14th Annual International Conference on Industrial Engineering and Operations Management
Dubai United Arab Emirates (UAE), February 12-14, 2024
Publisher: IEOM Society International, USA
Published: February 12, 2024
DOI: 10.46254/AN14.20240089

Quality Assurance Machine for Software Using Machine Learning

Shane Downing and M. Affan Badar
Bailey College of Engineering and Technology
Indiana State University
Terre Haute, IN 47809, USA
sdowning8@sycamores.indstate.edu, M.Affan.Badar@indstate.edu

Christopher J. Kluse
Faculty of College of Technology, Architecture and Applied Engineering
Bowling Green State University
Bowling Green, OH 43403 USA
ckluse@bgsu.edu

Abstract

Extending the work of Downing & Badar (2022), this paper presents the history of software, artificial intelligence, software quality assurance, and a software QA architecture called the Quality Assurance Machine (QAM). Using Design Science Research (DSR), the QAM was extended to support software containing ML models. Using descriptive statistics and hypothesis testing, this paper answers the research question: is QAM effective for assuring the quality of software containing ML? Future DSR efforts will add a Process Quality engine to the QAM to monitor and improve the processes used to create software and models.

Keywords
Quality assurance, traditional software, machine learning, design science, hypothesis testing

1. Introduction
Downing & Badar (2022) in their paper on holistic QA wrote: "The software industry is in transition: The traditional space, software 1.0, has seen more than fifty years of creation, testing, and delivery of deterministic software, but this tradition is being disrupted by the stochastic model of software 2.0." The paper (Downing & Badar, 2022), published in the summer of 2022, examined the possibility of using traditional quality assurance methodologies (software 1.0) for machine learning applications (software 2.0), and on November 30th of that year the OpenAI large language model, ChatGPT, was released to the public. Reaching 100 million monthly users two months after launch, it became the fastest-growing consumer application in history (Hu & Hu, 2023). Disrupted speaks to impact of a change, and it remains true that AI is having an impact on the software industry and industries that rely on software. There are businesses seeing no or low impact due to AI, examples being those related to human touch and interpersonal interaction (Ezell, 2019), requiring ethical decision-making (Mittelstadt et al., 2016), or necessitating emotional depth and empathy (Turkle, 2017). According to an IBM study, 68% of businesses in the United States plan to invest in AI by 2025; the non-AI businesses are either already, or soon to be in the minority.

Given the nature of the change AI is creating, perhaps disruption is too insufficient a label. Khun (1962) describes science as a process involving occasional revolutions followed by periods of normality where the details of a new paradigm are fleshed out, and Dhar (2003) notes a paradigm is a set of theories and methods accepted by the community. Over the course of its more than fifty-year history (Kneuper, 2017) software has seen advancements in

© IEOM Society International

how it is written (from assembly to high level languages), how it is designed (structured to object oriented), and how it is developed and deployed (waterfall to agile). However, none of these advances have risen to the level of having technology leaders and researchers urge a pause in their development due to profound risks to society and humanity (Metz & Schmidt, 2023). The AI disruption is special; the AI disruption is a paradigm shift, and industry may not be prepared.

Using a literature review to show the creation and evolution of software and its associated quality methodologies, and how that history can be used to support 2.0, this paper presents evidence that traditional Software Quality Assurance (SQA) can be effective for assuring product quality of software containing ML models. Beginning with an established SQA architecture, the Quality Assurance Machine (QAM), this paper also shows how the QAM was extended using Design Science Research to support ML, and as quality assurance is product and process quality concerns (IEEE, 2014), a process quality methodology is also proposed, thus making the QAM a holistic architecture that supports software, ML, product, and process quality.

New but Not Fundamentally Different

The first author of this paper works for an AI software company in the medical space. The machine learning (ML) models developed are not generative, that is, they do not create text responses to user input as large language models do. But ML models, like software, follow a common creation and deployment pattern: whether generating text responses, driving a car, or standardizing medical data, the ML pipeline is generally one of data engineering, model training, model deployment, and model maintenance (Ashmore et al., 2021). ML models have another thing in common, as shown in Holistic QA (Downing & Badar, 2022: there is no industry standard quality assurance methodology for software products containing ML, and using software to deploy ML models is a common strategy (ml-ops.org, 2023), as shown in Figure 1:

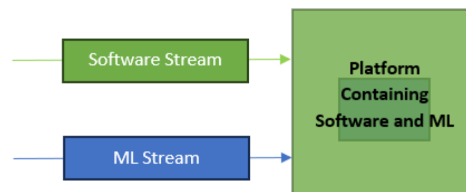

Figure 1. Platform Containing Traditional Software and ML Models

With the rapid adoption of AI solutions this creates a host of problems in industry, one of which is the potential duplication of quality assurance resources, or ad hoc solutions that rely on unqualified individuals (Downing & Badar, 2022).

The paper Holistic QA posed the research question “Can ML model QA be performed effectively using the same team and processes as that used for traditional software?”. An evaluation of the lifecycles of software and ML models determined there are enough similarities that the answer was “Yes, perhaps.”. The research in this paper shows the qualified answer for product quality is yes, but as QA is both product and process concerns, a design science research-based solution that supports process quality is also presented. Vom Brocke notes Design Science Research (DSR) is intended to extend the boundaries of human and organizational capabilities by designing new and innovative artifacts represented by constructs, models, methods, and instantiations (Vom Brocke, 2020), and Hevner presents DSR as a sequence of expert activities with a build-and-evaluate loop iterated a number of times before the final design artifact is generated (Hevner, 2004). As shown in this paper, the QAM was used to provide a qualified yes for product quality. This architecture has been used across multiple companies and has supported the lifecycles of firmware to cloud-based software. It has been iterated for software QA and has proven effective. After a year and a half of use in the artificial intelligence space, there is evidence to support its effectiveness for ML product quality. This is not a surprising finding as the software lifecycle has not changed dramatically in decades (Kneuper, 2017), parts of the ML lifecycle are essentially the same as for software, and most importantly, ML is software (Alamin & Uddin, 2021). This paper answers the product quality portion of the research question: Can quality assurance be performed effectively for ML solutions using the same process as that for software? Process quality, due to its socio-technical and cross-functional nature, requires a longer build-and-evaluate loop before a complete answer to the question can be given.

© IEOM Society International

2. Historical Background

2.1 History of Software

The first software appeared in the 1940s and was closely tied to hardware development with original languages being machine and assembly (Kneuper, 2017). The 1950s saw the appearance of high-level languages like FORTRAN and COBOL, and the 60s, a period when the software sector was unable meet the demands of the growing computer industry, became known as the software crisis (Dijkstra, 1972). The crisis, stemming from a lack of standardized tools, methods, and processes, was resolved through improved software design and formal development processes. Structured programming, using languages such as C invented by Dennis Ritchie at Bell Labs (Ritchie, 1978), assisted in resolving the crisis, but software design and languages continued to evolve, and it is difficult to see the history of early languages in today's designs. Software processes also appeared in the 70s and helped in resolving the crisis, but unlike designs and languages that were eventually replaced, they formed the foundation of processes in current use. The Waterfall model, while less common today, is a foundational software development process that introduced the phases of Requirements, Design, Development, Testing, and Release (RDDTR) that are still used in today's more agile methodologies (Kneuper, 2017); RDDTR is an important reason why traditional SQA can be effective for ML products. The 50s through the 70s was a period where the software lifecycle began gaining independence, moving away from playing second fiddle to hardware, and the changes that took place in the 80s and 90s set the stage for software now having a 29% larger market valuation than the hardware industry (IDC, 2023).

As with Waterfall's RDDTR, the 80s brought a software change that remains to this day: Object Oriented Design (OOD), beginning in the 60s and 70s with Simula and Smalltalk but not popularized until the introduction of languages such as C++, facilitated the creation of reusable and maintainable software (Stroustrup, 1986). Using objects and classes versus procedures, OOD is a natural way to model the real world, thus allowing software to represent things people understand: a car, a customer, a cup of tea. New designs have since emerged, such as Clojure (Hickey, 2008) and Reactive (reactivemanifesto.org, 2014), but these have complimented versus replaced OOD. It is estimated that 80% of software organizations use OOD in some form (OMG, 2018); this implies that much of the internet, search engines, smart phones, and mobile apps, the advancements of the last few decades, have been made using OOD. From a process perspective Waterfall was falling out of favor in the 80s due to its inflexible nature and replacements appeared, such as spiral which introduced iterative principles and allowed for flexibility and adaptation to changing requirements (Boehm, 1986). However, it was the 90s experimentation with agile methods and the 2001 publication of the Agile Manifesto (Beck et al., 2001) that would give rise to 71% of software projects today using agile methodologies such as Scrum, Extreme programming, and Lean (Statista, 2023).

2.2 History of AI and ML

Perhaps surprisingly, the history of artificial intelligence is similar in length of time to software, the first software appearing in the 1940s (Kneuper, 2017) as is the case for the first AI research (Haenlein, M., & Kaplan, A., 2019). However, while software grew continuously and its history shows progress through the decades, AI went through what some classify as seasons (Haenlein & Kaplan, 2019). The AI spring and summer, its beginning and expansion, lasted until the early 1970s when some in the research community questioned the future of AI and the United States congress decreased its funding. Thus began the AI winter. Haenlein & Kaplan note a reason for winter was the technology used, early attempts at AI being Expert Systems which are collections of rules instantiated as if-then statements. While these systems work well in areas that can be formalized, they perform poorly in environments that require learning from data, such as facial recognition, prediction, and medical diagnostics. AI is growing rapidly today partly because the current approach works with an application's surrounding environment, using statistical methods based on its data, versus trying to force the environment to fit the application. This is not a novel idea, researchers proposed statistical methods in the AI spring but abandoned the attempt when it was discovered processing power and data for training were insufficient (Haenlein & Kaplan, 2019). Today, processing power and available data to train models are no longer lacking, and winter is over.

2.3 History of Software Quality

The history of software quality is similar to the history of software. In the 1940s and 50s the concept of quality did not exist in the software industry, and this was a contributor to the software crisis of the 60s. As noted in the history of software, designs and languages were improved from the 60s through the 80s as were the processes used to create and deliver software. But while industry specific methodologies were developed, like Waterfall, the foundational principles of quality came from theories and practices from leaders such as Juran, Deming, and Crosby. Juran defined

© IEOM Society International

quality as fitness for use (Juran, 1999), Deming paraphrased quality as defined by the customer's needs and indicated good quality means a predictable degree of uniformity and dependability with a quality standard suited to the customer (Deming, 1986), and Crosby stated quality is conformance to requirements (Crosby, 1979). These definitions align with the IEEE definition used in the software industry: "Quality is the degree to which a system, component, or process meets specified requirements, and the degree to which it meets customer or user needs or expectations" (IEEE, 1990). While software quality can be broken down into multiple factors such as correctness, performance, reliability, and usability (Ashrafi, 2003), ultimately the quality of any product is measured by what the customer desires.

An interesting thing happened in 2001, precipitated by attempts in the 90s to resolve the inflexibility of Waterfall. A group of software developers created a document known as the Agile Manifesto and the impact it had on the software industry was revolutionary, agile development methodologies now being the most popular in the world (Statista, 2023). Martin Fowler, in Agile Manifesto: 20 Years Later, states the manifesto was a success but issues remain. The lack of a clear definition of agile has led to multiple agile methodologies, and agile is difficult to scale to large and complex organizations (Fowler, 2021). Fowler's conclusions are empirical, as are the following: Agile uses Lean, Kanban, and retrospectives to continually improve software development processes (Derby & Larsen, 2006). An observation by this paper's first author is that while those concepts exist in agile, they are only occasionally practiced and accomplish little with respect to process quality improvement. As an empirical example, scrum was used in four companies and six different software teams, and each team included a retrospective at the end of a time-boxed period of development known as a sprint. No team had a systematic way of acting on findings, and retrospectives were merely an opportunity to discuss shared difficulties. This has inherent value, but it does not make processes more efficient. QA is both product and process quality concerns. Software product quality, with quality definitions firmly rooted in customer expectations, has stood the test of time and the agile revolution. Process quality seemingly has faded into obscurity and critics are met with quotes from the manifesto: "People over process", and "Responding to change over following a plan". An empirical observation is agile happened not because it needed to, but because it could. The hardware industry continues to use Waterfall and has not withered and died, but it too is now exploring the benefits of agile (Huang & Boehm, 2014). Waterfall lasted so long because there was no better alternative for a product that is difficult to modify late in the development cycle. Software is flexible, and perhaps the creative people who struggled with being stifled by process saw this and took advantage? Today, software product quality remains in an organization's conscience because a remnant of Waterfall made it through the revolution, and this paper proposes a methodology to bring process quality back to its rightful station, one of being equal to product.

2.4 History of The Quality Assurance Machine (QAM)

Quality is a simple concept and several definitions have been noted, but while the definition of quality is simple, the quality lifecycle can be a fuzzy, abstract journey. As seen from the definitions of quality and quality assurance, there is room for subjectivity. Pascale and Anthos note that the inherent preferences of organizations are clarity, certainty, and perfection. Organizations strive for reliable and predictable outcomes (Pascale & Anthos, 1981). Pascale and Anthos also note that the inherent nature of human relationships involves ambiguity, uncertainty, and imperfection. Quality and quality assurance are product and process concerns managed within a socio-technical system, and the subjective nature can transform what should be objective assessments into a muddled space with unexpected factors, such as packed phrases and politics. Quality and quality assurance are simple concepts but can be confounding in practice, and effective quality solutions should consider product and process with an understanding that both come with ambiguity, uncertainty, and imperfection.

Since the 1960s, software development and its associated quality activities have followed the general process flow of Requirements, Design, Develop, Test, and Release (RDDTR). There have been modifications to the internal workings of this flow, from waterfall to spiral to agile, but software creation continues to follow the RDDTR paradigm. This is not surprising, as the goals of delivering software are to meet specifications and user requirements, and the generic phases of RDDTR are focused on both. Containing forward-looking process and retrospective product concerns, quality assurance is a set of activities that uses audits, metrics, and tools to ensure software life cycle processes work to prevent defects (Pressman, 2016). Process requirements are risk mitigation efforts - a forward-looking process view of quality and are also necessary to ensure timely delivery of software products by ensuring efficiency and repeatability. SQA also determines if software products are of suitable quality for their intended purposes. Are there defects in the software that has already been constructed? This is a retrospective view of product quality. The International Organization for Standardization (ISO) defines a product as the result of a process, they are fundamentally connected (ISO, 2015). The study of Business Process Management states processes should be

© IEOM Society International

standardized, measurable, repeatable, and reusable (ASQ, 2021). In industry there are methodologies and tools that support process and product quality as independent entities: Lean six sigma at a company level and agile at a software team level are process concerns, and automated test frameworks such as Selenium and Robot are product focused. As will be shown, the Quality Assurance Machine supports both process and product quality by merging the disciplines of QA into a coherent body of theory and practice.

The QAM did not start as an SQA architecture, its initial focus being purely the creation and execution of automated tests, a product quality concern. Initially it was not called the QAM, it was simply a lightweight automated test framework based on Python, one of many on the landscape. In relating the history of the QAM a question arises: If there are industry standard options for testing software and firmware, why create one? The answers are varied, but ultimately come down to one: there is always a need to create innovative solutions to real-world problems, and in companies from 50 to 30,000 employees, there was always a product quality need not filled by off-the-shelf software, and process quality needs not met by the Software Development Life Cycle (SDLC).

Designed to address persistent quality assurance issues seen across multiple companies developing firmware to cloud applications, the design of the QAM is based on schema learning, which attempts to build mental connections between a well-known concept and something new (Tervooren, 2015). A reference model of a machine with engines encapsulating specific SQA activities (Figure 2) provides an image of the quality assurance function that is easily understood and moves the SQA focus from a team to a functional perspective.

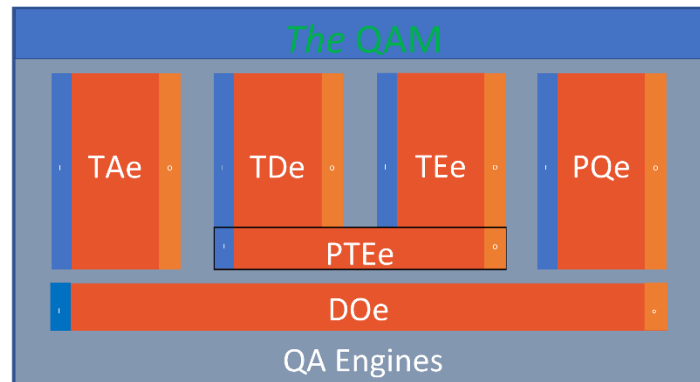

Figure 2. The Quality Assurance Machine (QAM)

Further exploiting the machine analogy, each engine performs process work: consuming inputs, adding value, and providing outputs (ASQ, n.d.). The QAM contains the following product and process QA engines:

QAM Product Quality Engines
TAe (Test Analysis engine) analyzes software release deltas and creates QA plans
TDe (Test Development engine) develops test coverage to support QA plans
TEe (Test Execution engine) executes automated tests and logs errors
PTEe (Python Test Execution engine) provides test automation to the TDe and TEe engines

QAM Process Quality Engines
PQe (Process Quality engine) process improvement engine using the PDCA workflow
DOe (DevOps engine) provides QAM process automation

There is always a need to create innovative solutions to real-world problems. This statement is part of the definition of Design Science Research (DSR), and DSR was informally used to build and modify the QAM as it progressed from an automated test framework to the SQA architecture in Figure 2. DSR is now being formally used to extend the QAM to support ML and to address the process quality gap seen in organizations using agile.

© IEOM Society International

3. Methodology

The purpose of this study was to extend a traditional software quality assurance architecture to support traditional software and ML model quality assurance simultaneously. The methodology used has three components: 1) Design Science Research to extend the QAM, 2) an analysis of the QAM's effectiveness in assessing product quality, and 3) a process improvement proposal to address the process quality gap seen in organizations using agile.

3.1 Design Science Research (DSR)

DSR seeks to enhance human knowledge with the creation of innovative artifacts and the generation of design knowledge via innovative solutions to real-world problems (Hevner et al., 2004). DSR is differentiated from a case study which provides deep insights into specific instances or phenomena (Yin, 2009). Cloutier and Renard note that the publication of The Sciences of the Artificial by Herbert Simon in 1969 marked a major turning point for research in management and administrative sciences (Cloutier & Renard, 2018). They indicate Simon's work has highlighted the importance of human activity at the heart of the design of artifacts to achieve goals and shape the world around us. However, while notably important in many fields, DSR has been met with skepticism within the research community. Peffers recalled a comment made by the editor-in-chief of a leading journal regarding submission of a paper based on DSR: "The journal didn't entertain papers about new systems development methods, because they involved neither theory development nor theory testing" (Peffers, 2020). The empirical gap in the literature discovered by Downing and Badar (2022) is an example of the need for a methodology such as DSR; research into the impacts of artificial intelligence on industry has not kept pace with the adoption of AI, and industry is not waiting. The research-practice gap is increasing, and methodologies such as DSR can be used to create rigorous and relevant artifacts as solutions to real-world problems.

DSR has seven guidelines as shown in Table 1 (Hevner et al., 2004). Each of these guidelines will be discussed relative to extending the QAM to support ML models.

Table 1. Design Science Research Guidelines

1. Design as an Artifact	Design-science research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation
2. Problem Relevance	The objective of design-science research is to develop technology-based solutions to important and relevant business problems
3. Design Evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation methods
4. Research Contributions	Effective design-science research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies
5. Research Rigor	Design-science research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact
6. Design as a Search Process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment
7. Communication of Research	Design-science research must be presented effectively both to technology-oriented as well as management-oriented audiences

1. Design as an Artifact

The QAM is an architecture, and once constructed is an instantiation. The QAM fulfills guideline 1.

2. Problem Relevance

As noted in the introduction of this paper, AI is a paradigm shift impacting technology on a global scale. The research conducted by Downing & Badar (2022) indicate industry may not be prepared for this shift. The QAM attempts to address part of the issues industry faces.

© IEOM Society International

3. Design Evaluation
Hevner offers various design evaluation methods, one of which is constructing scenarios to demonstrate artifact utility (Hevner et al., 2004). QAM design evaluation from a product quality perspective was accomplished with real-time scenarios, the QAM being used to evaluate product development and production releases.

4. Research Contributions
Extending the QAM to simultaneously support products containing software and ML models fills the gap discovered in the paper Holistic QA (Downing & Badar, 2022).

5. Research Rigor
Hevner notes that rigor must be assessed with respect to the applicability and generalizability of the artifact, and an overemphasis on rigor can lessen relevance (Hevner et al., 2004). There are many ways to ensure rigor in DSR; how those were performed for extending the QAM is discussed:
Problem Definition: The problem and its relevance were clearly defined in Holistic QA (Downing & Badar) and in this paper.

Theoretical Foundation: The QAM architecture is based on the SDLC phases Requirements, Design, Development, Test, and Release. This is a body of practical and theoretical knowledge developed over many decades.

Artifact Design and Construction: The QAM was constructed in a structured, iterative, and intentional manner. Instantiated designs modified as new information was discovered.

Evaluation: The QAM was constructed, executed, and its performance evaluated daily during product development activities over a nearly two-year period. Evaluation criteria were built into the normal process workflow of creating and releasing software and ML models.

Knowledge Contribution: Extending the QAM to simultaneously support products containing software and ML models adds to body of knowledge by filling the gap discovered in the paper Holistic QA (Downing & Badar, 2022).

Transparency, Replicability, and Generalization: The paper Holistic QA has been published and this paper will be submitted for publishing. The methods used to construct the QAM, documented in this paper, are standard software engineering practices. The QAM has been used multiple times with varying product technologies. It is believed the QAM extended to support ML can be generalized to other products that are a mix of software and ML models.

6. Design as a Search Process
As noted, a period of evaluation different SQA frameworks occurred prior to selecting the QAM architecture as the correct choice for the specific environment.

7. Communication of Research
The paper Holistic QA has been published and this paper will be submitted for publishing.

3.2.1 Extending the QAM to Support ML
The history of the QAM is one of design science, creating innovative artifacts to solve real-world problems. Initially the QAM contained only two of its current engines, TDe and TEe, and the automated test framework PTEe, the original intent being only to support product. Over the course of multiple companies requiring unique solutions, the original architecture remained, which is why the QAM is considered an architecture versus a design, architecture being "what" is being created and design being "how" it is created. The architecture, which has previously been known as FTF, CTC, and KAYAK, was named the QAM when the scope of "what" changed from product quality (software testing) to product *and* process quality (quality assurance). Additional components were added to the architecture, a reference model was adopted (a machine), the machine was populated with independent processes to perform the work of quality assurance (engines), and it was given the name QAM.

Today, the QAM is quite different from its original instantiation, as is much of the software industry with the introduction of AI. To construct today's QAM, DSR was used to provide scientific rigor and relevance: rigor to what were previously solutions in the moment, and relevance by creating an artifact which fills an empirical gap. The QAM

© IEOM Society International

artifact is now a quality assurance architecture supporting software and ML model QA simultaneously. Figure 3 is a visual representation of why it was believed traditional SQA could be used to support platforms containing ML - in the context of RDDTR, they are essentially the same.

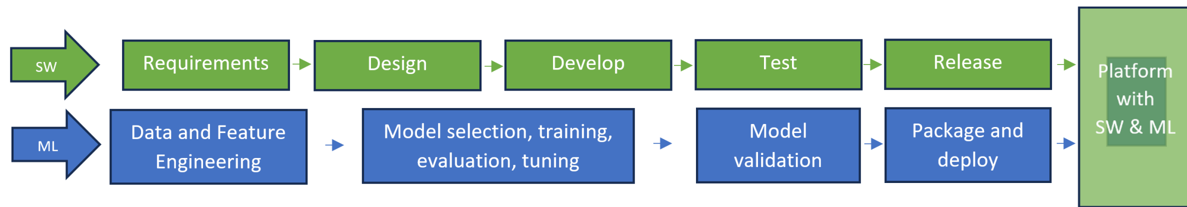

Figure 3. Concurrent Software and ML Development

ML models have requirements (R), they must be designed and developed (DD), and finally tested and released (TR). The terminology and methods used are very different in the ML pipeline compared to software, but the result is the same: a software product that must meet requirements and be successfully deployed. From an SQA perspective, the ML flow is similar to traditional software, however, under the hood of this high-level depiction is where the technologies diverge. ML models are data-driven and require clean and labeled data to train models and achieve accuracies acceptable for production deployment. While traditional software development does provide interim builds for testing in the development phase, ML model construction remains firmly in the purview of data science teams due to the experimental nature of model training, probabilistic model outcomes, and skill sets not in the scope of software engineering. As discussed in the results sections, this speaks directly to why there is a qualified yes to the research question this paper answers.

Kästner notes quality assurance for systems containing ML needs to be planned holistically, and that evaluating accuracy of the ML model is only one step. Integration and system testing is necessary to verify correct behavior of the interactions of all platform components, which includes ML (Kästner, 2020). The QAM that supports software and models does just this. Using DSR, test engines specific to models were added, as well as product classes that when instantiated represent ML models. The PTEe framework was also modified to instantiate and test against only those ML models present in the executing platform. The ML-extended version of the QAM contains tests for software, tests for models, tests for software that interact with models, and framework and library support for the mixed technology system, all within a single executing QAM instance.

3.2 Software and ML Product Quality

The History of Software discussed how SDLC methodologies have changed over the decades, but the theme of the change has been one of increased adaptability, flexibility, iteration, collaboration, and continuous feedback. There is not an SDLC methodology in widespread use that does not follow RDDTR. This is foundational to software product quality efforts, and as the QAM's architecture and associated processes align with RDDTR, the QAM rests firmly on a foundation of software engineering theory and practice. The usual starting point for a software test or SQA organization is verification of software requirements. As infrastructure and processes mature, QA coverage expands to other attributes, such as performance and reliability. Figure 3 shows the same QA opportunities exist for ML models, but traditional software testing methods are not effective for all models concerns, such as verifying accuracy. However, the ML pipeline aligns with software and models do have functional requirements that can be verified using software test techniques. This makes using traditional software test methodologies (analyze requirements, create tests cases and automated tests, execute tests to verify conformance to requirements, report results) worthwhile for verifying model quality also.

3.3 Adding the Process Quality engine (PQe)

The motivation for the PQe comes from several personal observations and from published research findings. The first personal observation, discussed in the History of Software Quality, is that the agile methodology contains mechanisms for process improvement, such as retrospectives. These are either rarely used, or if used, action on outcomes is rarely taken. A second observation is that agile teams typically use project management software designed to manage development and QA activity using workflows, and these workflows are strictly followed. A final observation is agile teams typically meet daily for a short meeting known as a stand up to synchronize their work and plan for the next 24 hours. These observations create an opportunity for a novel process improvement methodology managed in the same

© IEOM Society International

time and space as all development and QA efforts, using stand ups and workflows. Figure 4 shows an example workflow for a Jira Story, Jira being a common agile management tool (Atlassian, n.d.):

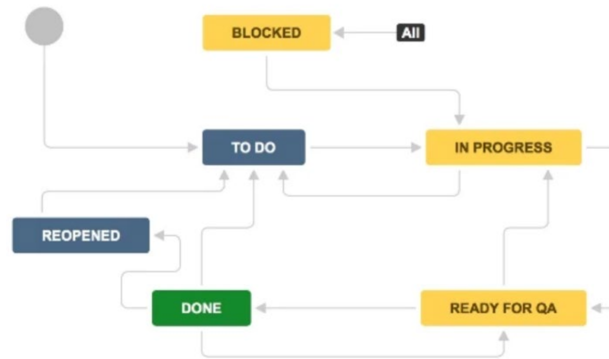

Figure 4. Jira Story Workflow

This workflow and information in the Story tell the history of its life: who created it, who owns it, where it is in its cycle, and what other items may be associated with it. The workflow is consulted often to understand where something came from and where it is going. The PQe will take advantage of this consistent approach to managing agile work using a Process Improvement (PI) type with a Plan Do Check Act (PDCA) workflow. A PI item will be created when process issues are discovered, someone within the organization (SQA, Engineering, Data Science, Product, etc.) will own the item and move it through the PDCA phases. The PI will be moved to the DONE state when all PDCA phases have been completed. Documentation of the improvement will be contained in the PI issue. The PQe will be responsible for monitoring the process quality metrics of effectiveness, efficiency, and equability for processes within the software and ML delivery streams. In this way SQA, using the QAM, can monitor and act on the quality of the processes creating software and models as well as the quality of the actual products, thus becoming a complete QA solution.

The research finding is from the Hevner paper on DSR where he notes it is incumbent upon researchers to “further knowledge that aids in the productive application of information technology to human organizations and their management", but doing so involves the complementary but distinct paradigms of behavioral and design science (Hevner, 2004). Hevner states that creating artifacts is an opportunity to make significant contributions by engaging in the complementary research cycle between design science and behavioral science, as they are not dichotomous in an information system. Basically, it takes people to create an artifact, and understanding how to manage the people part is as important as the artifact part. This reality, it takes people to deliver a product, is the main motivator behind the “same time and space” PI workflow using agile. People are accustomed to this workflow, people communicate the issues they are having daily, and it requires people to resolve them. The current agile PI solutions are not in the same time and space as the product creation activity, and they are ineffective.

4. Results

There are two areas of results to be considered for this effort: 1) The result of building the QAM using Design Science, and 2) the result of using the QAM to assure product quality and answering the question is it effective for software and ML model QA.

4.1 Building The QAM

Construction of the QAM began in the fall of 2021 with the first artifacts completed in December. Prior to constructing the QAM other solutions were investigated: the Robot Framework was considered (Pekka Klärck, 2020), as well as the Go programming language (Donovan & Kernighan, 2015). These are but two examples of many product quality solutions, but they support only half of QA’s scope. Figure 5 shows the progression of QAM construction over a nearly two-year period, the vertical axis being the progression of items entering each QAM engine. Engine items, sub-artifacts from a DSR perspective, are a combination of Bash and Python scripts, Python modules, and text-based configuration files. The choice of Bash and Python was more an environmental concern than a QAM requirement. The QAM is an architecture and defines the shape of the solution; the design of the QAM, how it is constructed, is

© IEOM Society International

driven by the product environment it will support. The series lines in Figure 5 speak to nearly two years of design and iterate by a Software Quality Assurance organization. The story of that effort is discussed below.

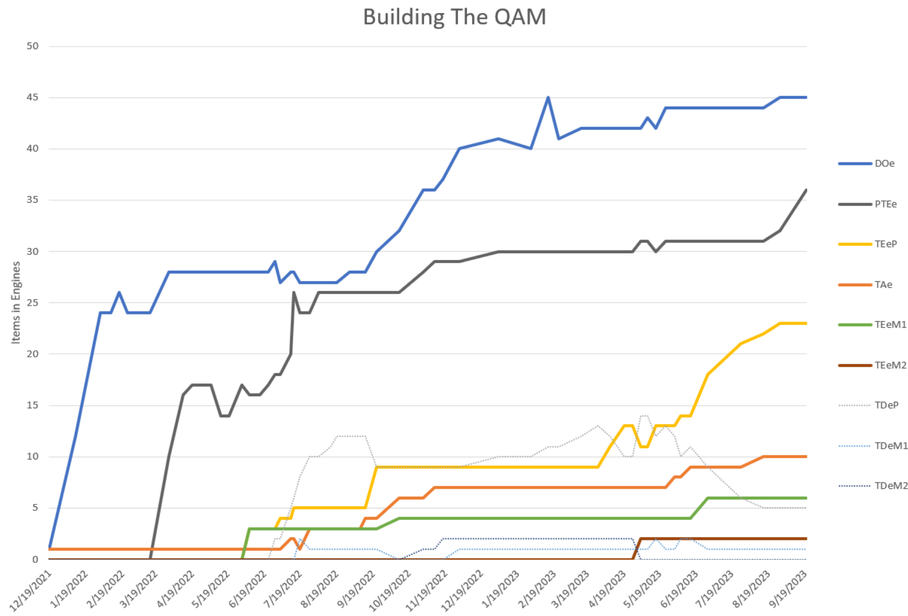

Figure 5. Building The QAM

4.1.1 DOe Construction

It would be odd for a software company to delay product development until a testing solution is in place; the experience of this paper's first author is that months, even years of development will have taken place prior to a test or QA organization even being considered. The DOe, a collection of artifacts to automate all processes that can and should be automated, was created first because product development does not wait, and until automated tests are available, manual testing must be performed. The first items to enter the DOe were Bash scripts which facilitated a standardized launch of the platform and associated ML models, an important activity for a multi-component system. The DOe, by automating the build and release steps, also inherently monitors these processes for unexpected changes, a key contributor to the overall process quality of the product lifecycle, and one that will be formalized with the addition of the PQe. As an indicator of DSR being used to build the QAM (build, evaluate, iterate), Figure 5 shows that 55% of current DOe functionality was created prior to any automated test framework items (PTEe). Months of effort went into designing and iterating on an artifact that was stable and met system-level QA requirements prior to designing and building the remaining QAM engines.

4.1.2 PTEe Construction

The Python Test Execution engine provides a foundation on which software testers can build tests faster, more efficiently, and that require less maintenance as the software being tested evolves. In other words, it is an automated test framework (Fewster & Graham, 1999). PTEe provides a foundation to enable a common look, feel, and execution flow for all QAM tests, whether they are tests against the platform software, standalone ML models, or both. PTEe also provides a common set of command line options available to all tests, and common error handling and logging methods. Common is key as this enables features such as reusability (PTEe has a common library utilized by all tests – write once and share), maintainability (changes can be made in the framework versus every individual test), consistency (all tests have a common structure and output), and integration (PTEe enables compatibility with the DOe and non-QAM software). Figure 6 represents the interaction of tests in the test engines with PTEe and ultimately the product under test. An important point to note is that when the system in Figure 6 is executing and interacting with the Product Under Test (PUT), it is communicating with the platform software and the ML models in the same execution. There are not two automated test ecosystems, there are not two SQA teams. There is one SQA automation solution, one SQA team, and one SQA process for both software and ML models.

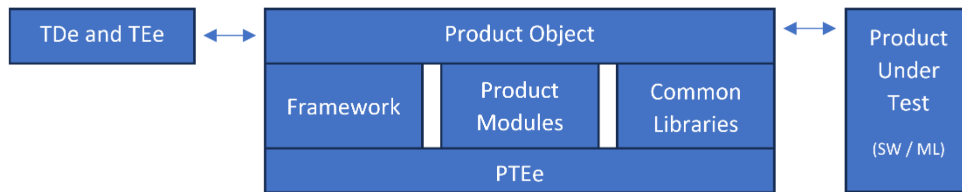

Figure 6. Automated Test Ecosystem

PTEe Framework

The benefits of a framework cannot be overstated. As an empirical example, prior to PTEe and the test engines coming online, several ad hoc automated tests against product requirements were developed, partly as a learning exercise but also to use for future testing. More than one hundred hours were spent creating these tests, eventually followed by a refactoring effort to align them with the current common test model. Refactoring was necessary for two reasons: 1) each ad hoc test was a unique instance reflecting a particular developer's preferences, thus making the task of deciphering and maintenance difficult as there was no common context within the test population, and 2) there was no common library support at the time, therefore each test contained unique ways of managing data and interfacing with the test target; this necessitated test-level changes for items that are not test-level concerns. A test is defined as the process of evaluating a system or component by providing specified inputs, observing the results, and comparing the observed results to the specified expected results (IEEE, 1990). Inputs, results, compare; a test should only test, and this is precisely what all TDe and TEe tests are designed to do. Anything beyond that is a PTEe concern.

PTEe Product Modules

Automated tests and the content in PTEe follow an Object-Oriented design. As mentioned, using objects versus procedures is a natural way to model the real-world using software, and PTEe contains Python modules which represent products. As an example, the software platform discussed in this paper is contained in a module named platform.py, and when instantiated becomes an executable object that interfaces with the actual PUT. This abstraction allows tests to be written as if communicating with the executing product; the PUT can be interrogated for version information (platform.get_version()) or it can be instructed to perform a workflow (platform.execute_workflow(workflow_a)).

PTEe Common Libraries

The common library is composed of Python modules that contain common functionality to interface with products, manage data, or execute tasks performed often. Using the previous example of a workflow, requesting an action be performed is also an example of common library functionality, as the request to execute a particular workflow utilizes common functionality that is not specific to the platform object. In this way any instantiated product object (platform, ML model, etc.) has access to common functionality, and that functionality has a single point of maintenance.

4.1.3 TDe and TEe Construction

The Test Development and Test Execution engines are simply containers for automated tests, TDe being the place a test is created and TEe where it moves when proven effective and efficient. To perform system regression testing, PTEe contains a test runner that executes all TEe tests, platform and model, against all new releases of the platform and associated models. The dashed lines in Figure 5 represent the development engines in the QAM, and the fact that they do not trend up is by design; the TEe lines increase by the same amount TDe's are decreasing as stable tests move from one engine to the other. Figure 5 details the number of tests for software (TEeP) and models (TEeM1 and TEeM2). There are considerably more tests for software, and this is partially due to the nature of testing a GUI-based platform and to the fact that the pace of model development, due to its data-driven pipeline and need for experimentation, can be longer than for software (Davenport & Patil, 2012).

4.1.3 TAe Construction

The Test Analysis engine has been used in various ways depending on the context. Previously, TAe contained

automation to analyze source code in new releases and identify gaps in test coverage or changes to sensitive areas, such as an API. Today TAe is used to monitor the software development process space, creating reports detailing gaps in requirements coverage and analyzing information contained in external files critical to regulatory compliance.

4.2 Product Quality

Software product quality has many views, but the attributes first evaluated most often are functional correctness and conformance to requirements. Descriptive statistics and hypothesis testing were used to arrive at an answer to this paper's research question.

4.2.1 Descriptive Statistics

It was stated that this paper presents evidence that traditional Software Quality Assurance (SQA) can be effective for assuring product quality of software containing ML models. There are many attributes associated with software quality, such as functional correctness and completeness, performance, security, etc. (IEEE, 1990). For model quality, the literature and industry are narrowly focused on attributes such as accuracy, interpretability, and fairness (Downing & Badar, 2022). However, as shown in Figure 3, there are many opportunities for model-related defects post model tuning. To show traditional SQA can be effective for ML models, a comparison of the functional correctness and completeness between software and models is presented. While still in the design phase, performance testing is being added to the scope of QAM functionality. The expectation is that model performance can be evaluated in a manner similar to software performance, thus further extending the QAM into the ML support space.

Table 2. Functional Correctness and Completeness Descriptive Statistics

	Software	ML Models
Automated Tests Created	27	10
Requirements Tested	83	26
Automated Coverage	42% (35 requirements)	77% (20 requirements)
Manual Coverage	58%	23%
Defects Discovered	157	56
Development Releases Tested	226	86
Production Releases Tested	9	6

Table 2 shows the results of a traditional Software Quality Assurance team using traditional SQA processes to test software and machine learning models. The SQA team that created the automated tests and performed the manual testing have a software engineering background, they are not members of a model development team, nor do they have model development experience. They are traditional software quality assurance engineers developing traditional software QA infrastructure and using traditional SQA processes to perform automated and manual testing of a software product containing ML models. Functional correctness and completeness are requirements verification activities, and while the numbers relative to requirements and releases are different for software and ML, these are driven by the demands of the business, not by the capabilities of SQA. In both cases there is 100% coverage of requirements, and this shows the SQA solution is effective for product quality relative to product correctness and completeness. Making observations based on descriptive statistics is useful, but a stronger statement of effectiveness comes from inferential statistics.

4.2.1 Hypothesis Testing for Defects Discovered

If an SQA team using the QAM and traditional SQA processes discovers the same number of defects in software as in ML models, it can be stated that the QAM is proven effective for assuring software and ML model product quality. To determine this, a hypothesis test using a two-sample z-test for proportions is used with a significance level alpha = 0.05, two-tailed.

The null hypothesis is that the proportions Requirements Tested to Defects Discovered for software and ML models are different:

H_0: $RT_{SW} / DD_{SW} \neq RT_{ML} / DD_{ML}$ => $83/157 \neq 26/56$

© IEOM Society International

The alternative hypothesis (H_a) is that the two proportions are the same:

H_a: $RT_{SW} / DD_{SW} = RT_{ML} / DD_{ML}$ => 83/157 = 26/56

For the two-sample z-test for proportions,

$n_1 = 157, x_1 = 83 \Rightarrow \hat{p}_1 = \frac{83}{157} = 0.529$

$n_2 = 56, x_2 = 26 \Rightarrow \hat{p}_2 = \frac{26}{56} = 0.464$

Pooled proportion $\hat{p} = \frac{83 + 26}{157 + 56} = 0.512$

$$z = \frac{0.529 - 0.464}{\sqrt{(0.512)(1 - 0.512)(\frac{1}{157} + \frac{1}{56})}} = 0.835$$

For a z score = 0.835 the p-value = 0.403, which is above the significance level set for this test. Therefore, the null is rejected and the alternative accepted. The two proportions are the same, and using the QAM for ML product quality is effective.

4.2.2 The Qualified Yes

This paper answers the product quality portion of the research question: Can quality assurance be performed effectively for ML solutions using the same process as that for software? Descriptive and inferential statistics have shown the answer to be yes, however, ML models have probabilistic outcomes whereas software is deterministic. The model pipeline, one of data engineering, model tuning, and accuracy verification, is very different from the SDLC, and at many points skill sets that are not software engineering are necessary to design, develop, test, and release an ML model into production. For example, in the medical space clinical expertise is required to verify model outcomes are clinically correct. This is not an SQA activity. Ultimately it becomes a question, perhaps another research question, of strategy. Is there more value derived by creating two teams using different processes and tools to perform QA of a mixed technology product? This paper does not attempt to answer that question, but it does prove that traditional SQA is effective for software containing ML models.

5. Conclusion

The first author of this paper has worked in the software industry for decades, and until two years ago knew as much about AI as was advertised in mass media. Even for a software professional, what AI was, the impact it was having, the impact it would have, was unknown. Twenty years ago, the Google search engine was like magic, and today it is slightly annoying, users having to piece together bits of information from multiple sites while negotiating more adds than relevant content. Large Language Models (LLM) like ChatGPT and Bard are similar to the Google magic of the past. Incredibly efficient and strangely effective, they are merely an example of where technology, not just the software industry, but technology, is heading. They have issues, and taking an LLM response as truth is risky business. That will change, and the change will likely be faster than anticipated. The software industry and the businesses that use software are experiencing a rapid paradigm shift, perhaps too rapid for the companies creating AI. The quality processes and teams needed to support the transition to AI are immature at best, ad hoc most likely. The literature does not support it, but from a quality perspective a possible bridge between the age of software and AI-enabled software is to lean into tradition. SQA solutions like the QAM are effective for assuring product quality in software containing probabilistic models, and follow-on research will attempt to prove they are also effective for improving the quality of the processes that create it.

This research did not directly address two concerns with products containing ML models, those being model maintenance post-production release, and approaches to shifting model testing left, that is, closer to the model development phase. ML models are trained on a labeled set of data, but it is not the entire population of data; it is a sample. In a production environment models process new data, and that data may change over time compared to the static training data. Financial data, weather data, population data, these are examples of changes in a data set. A model's response to this changing data can cause its accuracy to degrade, and this is known as model drift (Gama et al., 2014). Addressing model drift from a quality perspective is an opportunity for further research. Shifting the QA

activity left, closer to model development, is a planned activity and already part of the QAM's design. In the QAM, all models can be tested outside of the software where they run in production. The future challenge is to create the infrastructure necessary to build and deliver models outside of the normal ML pipeline.

References

Alamin, M. A. A., & Uddin, G. Quality Assurance Challenges for Machine Learning Software Applications During Software Development Life Cycle Phases. arXiv:2105.01195 [Cs], 2021, http://arxiv.org/abs/2105.01195

ASQ, *ASQ Certified Manager of Quality/Organizational Excellence Handbook, Fifth Edition*. Asq.org, 2021, https://asq.org/quality-press/display-item?item=H1569

Ashmore, R., Calinescu, R., & Paterson, C. Assuring the Machine Learning Lifecycle: Desiderata, Methods, and Challenges. *ACM Computing Surveys*, 54(5), 1–39. 2021, https://doi.org/10.1145/3453444

Ashrafi, N., The impact of software process improvement on quality: In theory and practice, *Information & Management*, vol. 40, no. 7, pp. 677–690, 2003, https://doi.org/10.1016/S0378-7206(02)00096-4.

Atlassian, JIRA: The #1 software development tool used by agile teams, Atlassian, n.d., https://www.atlassian.com/software/jira.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., & Kern, J., *Manifesto for agile software development*, 2001.

Boehm, B., A spiral model of software development and enhancement, *ACM SIGSOFT Software Engineering Notes*, vol. 11, no. 4, pp. 22-42, 1986.

Cloutier, M., & Renard, L. *Design Science Research: Issues, Debates and Contributions.* Projectics / Proyéctica / Projectique, 20(2), 11–16, 2018, https://doi.org/10.3917/proj.020.0011

Crosby, P. B., *Quality is free: The art of making quality certain*, New American Library, 1979.

Davenport, T. H., & Patil, D. J., Data Scientist: The Sexiest Job of the 21st Century, *Harvard Business Review*, 2012.

Deming, W. E., *Out of the crisis*, MIT center for advanced engineering study, 1986.

Derby, E., & Larsen, D. *Agile retrospectives: Making good teams great.* Pragmatic Bookshelf, 2006.

Dijkstra, E. W., *Structured Programming by Edsger Wybe Dijkstra*, 1972, https://www.abebooks.com/Structured-Programming-Edsger-Wybe-Dijkstra-Academic/31345549255/bd

Donovan, A. A., & Kernighan, B. W., The Go Programming Language, Addison-Wesley Professional, 2015.

Downing, S. and Badar, M.A., Holistic QA: Software quality assurance for the Machine Learning era, *Proceedings of the 7th North American Conf. on Industrial Engr and Operations Mgmt*, Orlando, USA, June 11-14, pp. 147-155, 2022, http://doi.org/10.46254/NA07.20220050.

Ezell, S. J., How Technology Is Reshaping the Concept of Play, *Issues in Science and Technology*, vol. 35, no. 3, pp. 35-43, 2019.

Fewster, M., & Graham, D., Software Test Automation: Effective Use of Test Execution, 1999.

Fowler, M., *Agile Manifesto: 20 Years Later*, Addison-Wesley Professional, 2021.

Hevner, March, Park, & Ram. *Design Science in Information Systems Research*. MIS Quarterly, 28(1), 75, 2004, https://doi.org/10.2307/25148625

Gama, J., Žliobaitė, I., Bifet, A., Pechenizkiy, M., & Bouchachia, A., A survey on concept drift adaptation, *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 1-37, 2014.

Haenlein, M., & Kaplan, A. A Brief History of Artificial Intelligence: On the Past, Present, and Future of Artificial Intelligence. *California Management Review*, 61(4), 5–14, 2015, https://doi.org/10.1177/0008125619864925

Hickey, R., The Clojure programming language, *Proceedings of the 2008 symposium on Dynamic languages*, p. 1, 2008.

Huang, L., & Boehm, B., Determining how much software is enough? A value-based approach, *Journal of Systems and Software*, 2014.

IBM Institute for Business Value, AI adoption 2022: AI is poised for hypergrowth as enterprises accelerate their AI journeys, 2022.

IEEE, IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990), IEEE, 1990.

IEEE, IEEE Standard for Software Quality Assurance Processes, IEEE Std 730-2014, pp. 1–138, June 13, 2014.

International Data Corporation (IDC), Worldwide Software Market Forecast 2021-2025, 2023.

International Organization for Standardization, ISO 9000:2015 Quality management systems — Fundamentals and vocabulary, ISO, 2015.

Juran, J. M., & Godfrey, A. B., *Juran's quality handbook (5th ed.)*, McGraw-Hill, 1999.

Kästner, C., On the process for building software with ML components, Medium, November 1, 2020, https://ckaestne.medium.com/on-the-process-for-building-software-with-ml-components-c54bdb86db24.

© IEOM Society International

Klärck, P., Robot Framework User Guide, Robot Framework Foundation, 2020.
Kneuper, R. Sixty Years of Software Development Life Cycle Models. 14, 2017.
Mittelstadt, B. D., Allo, P., Taddeo, M., Wachter, S., & Floridi, L., The ethics of algorithms: Mapping the debate, *Big Data & Society*, vol. 3, no. 2, 2053951716679679, 2016.
Metz, C., & Schmidt, G., Elon Musk and Others Call for Pause on A.I., Citing 'Profound Risks to Society.', The New York Times. March 29, 2023, https://www.nytimes.com/2023/03/29/technology/ai-artificial-intelligence-musk-risks.html
Ml-ops.org, 2023, August 11, Retrieved September 1, 2023, from https://ml-ops.org.
Object Management Group (OMG), The State of Object-Oriented Technology Report, 2018.
Pascale, R. T., & Athos, A. G. (1981). *The art of Japanese management*. Business Horizons, 24(6), 83–85. https://doi.org/10.1016/0007-6813(81)90032-X
Peffers, K., Tuunanen, T., Gengler, C. E., Rossi, M., Hui, W., Virtanen, V., & Bragge, J. Design Science Research Process: A Model for Producing and Presenting Information Systems Research (arXiv:2006.02763). arXiv, 2020, https://doi.org/10.48550/arXiv.2006.02763
Pressman, R. S., *Software Quality Assurance: A Practitioner's Handbook, Sixth Edition*, McGraw-Hill Education, page 1, 2016.
Ritchie, D. M., The C programming language, Prentice-Hall, 1978.
Statista, What are the most popular software development methodologies?, 2023.
Stroustrup, B., The C++ programming language, Addison-Wesley, 1986.
The Reactive Manifesto, 2014, Retrieved from https://www.reactivemanifesto.org.
Turkle, S., *Alone together: Why we expect more from technology and less from each other*, Basic books, 2017.
Vom Brocke, J., Hevner, A., & Maedche, A. Introduction to Design Science Research. In J. vom Brocke, A. Hevner, & A. Maedche (Eds.), *Design Science Research*. Cases (pp. 1–13). Springer International Publishing, 2020, https://doi.org/10.1007/978-3-030-46781-4_1
Yin, R. K., *Case study research: Design and methods (4th ed.)*, Los Angeles: Sage publications, 2009.

Biographies

Shane Downing is a PhD student at Indiana State University and a software quality assurance manager with more than two decades experience in the software industry. He has contributed to the software quality space as a test engineer, test architect, and manager. He received BSc degrees in Automated Systems Engineering Technology and Computer Science, an ME in Embedded Systems Engineering, and is an ASQ Certified Manager of Quality & Organizational Excellence (CMQ/OE).

M. Affan Badar, PhD, CPEM, IEOM Fellow is Professor and Director of PhD in Technology Management in Bailey College of Engineering and Technology at Indiana State University, USA. From 2016 to 2018, he was Professor and Chair of the Department of Industrial Engineering and Engineering Management at the University of Sharjah, UAE. He received a Ph.D. in Industrial Engineering from the University of Oklahoma, MS in Mechanical Engineering from King Fahd University of Petroleum and Minerals, MSc in Industrial Engineering and BSc (Hons) in Mechanical Engineering from Aligarh Muslim University.

Christopher J. Kluse, Ph.D., is the program coordinator and Associate Professor of Quality Systems at Bowling Green State University, USA. Prior to his full-time appointment in academia, he was a lecturer of Quality Management at Eastern Michigan University and an adjunct instructor for Southern New Hampshire University. Dr. Kluse was employed full-time for 25 years in the automotive industry, holding various quality management-related positions. He holds a Ph.D. in Technology Management and is an ASQ Certified Manager of Quality & Organizational Excellence (CMQ/OE).

© IEOM Society International