Software Architect as an Agent for Product Success

Brian Walker

walkerb@mac.com

Abstract

As software development projects become larger and more complex the need for a development roadmap becomes imperative. A software architecture provides the necessary roadmap to guide development and provides a high-level view of the system design and construction. The role of software architect provides a vital influence for the quality of a system. As guardian of the architecture, the architect can be a champion for software quality within a development team.

Biography

Brian Walker is Software Architect for the Video Product Line at Tektronix developing professional video monitors for television stations, cable operators and video content producers around the world. For the past 19 years, he has been the lead architect for a software architecture that started its life running on a real-time, embedded PowerPC processor within the first WVR video monitor product and has evolved through time to run on full desktop Linux system with a brief foray into embedded ARM processors. During that time, the architecture survived memory-constrained systems, handheld products and the transition from a big-endian to little-endian hardware architecture and evolved to meet the needs of current shipping products.

1 Introduction

Good design leads to quality. Although there are many paths that one can follow to realize a quality product, the one constant is that quality product follows from a good design. Whether that design derives from a top-down development pattern where a carefully constructed design is faithfully implemented or from a. Test Driven Development pattern where a design is discovered through frequent refactoring, quality follows design.

As projects become larger and more complex, the effort to design a product grows in complexity as well. Whereas a small system could simply evolve over time and their function could be easily understood by a single developer, complex systems defy the ability for a single person to intimately know the details of the entire system. No longer can the design of the system simply emerge after serious coding sessions or be briefly sketched on a napkin. As more developers are needed to construct a system, more people must become involved in the design of the system and their efforts must be coordinated and consistent. Large systems, therefore, need an architecture to define the overall design of the system, its parts and how those parts interact with each other to produce a unified whole.

The architecture requires the attention of a dedicated individual or team to create, manage, guide and protect the architecture to provide the necessary foundation, conceptual framework and roadmap for the development of a software system. An architecture may integrate existing products into a unified whole or it may establish the foundation for an entirely new system. The role of the architect is to assemble or create the architecture and communicate it to the team. Once development starts, the architect should guide the product development and champion the integrity of the architecture. The architect manages the evolution of the architecture to ensure that the architecture remains relevant and continues to serve the needs of the development team and happy customers.

2 Role of the Architect in an Organization

The architect is uniquely positioned to be a champion for the quality of a system. In a Scrum team, the participants typically have an area of interest that determines their focus.

The Product Owner is often focused on delivering value to the customer. The Product Owner might have technical expertise but often they bring the domain expertise of the customer to the team and must be able to understand and express the needs of the customer. Those needs are often expressed as features that the product will deliver to the customer and can be expressed by User Stories.

The Scrum Master works with the Product Owner to groom the backlog. The Scrum Master oversees and manages the Scrum process by which the product team delivers stories to the customers. Those stories tend to have an external focus since they are told from the perspective of the customer.

The Scrum team is charged with implementing the stories in each sprint. The team must add technical stories to define the infrastructure that is necessary to realize the user stories because they are the domain experts in the matters of product development. In the early stages of product development, the technical stories often outnumber the user stories. The team is often measured by their sprint velocity and by the amount of technical debt that they accumulate or remove during a sprint.

The interest for the architect is the architecture of the system and the architect’s focus is delivery of the architectural vision. The architect works with the Product Owner and the Scrum team to develop the architecture. The architecture provides the overall scope of the project and the basis for early story estimation in the form of T-shirt sizing. The architect may work with the Scrum Master and the Scrum team to define Epics which can be placed in the backlog.

While the effort to implement features can easily be measured in story points, matters of quality are hard to measure in terms of story points. Granted, defects can be used as a measure of technical debt or lack of quality, but, defects are a lagging indicator. Acceptance criteria allows teams to determine when the story has been completed so that it may be accepted by the Product Owner. Quality can be unknown until defects are discovered. A proactive approach to remove risk of defects in a system through a consistent architectural definition improves system quality by preventing defects.

Conformance to an architecture can be an indicator of system quality. Each unit of work should follow a design which is derived from the architecture. The architecture should define the interactions, the processes and interfaces between units such that units that adhere to that interface will interact correctly and reliably with others. An architect should review system design and measure how faithfully it follows the framework defined by the architecture.

As the guardian of the architecture, the architect can be a champion of quality. With a focus of architectural integrity, the architect promotes consistency with the software. Consistency reduces variations that produce opportunities for defects and increase the cost of testing. The architecture defines common infrastructure which is reused within the product. Through consistency and reuse, the architecture contributes complete and tested components that increase system stability.

The architect’s job is not complete when the architecture is finished. The architect’s roles transitions from defining the architecture to guiding the implementation of the architecture through product development. The value of an architecture is determined by its reuse, its adoption and its development beyond the needs of its first product.

3 Design of a Large System

3.1 Nature of Human Design

For a small, discrete system, one can discover the design through trial and error and repeated refactoring. The process quickly becomes unwieldy in larger systems. Where multiple systems interconnect, the job of coordinating the system requires a conceptual vision to explain the construction and function of the system as a whole. People specialize because it is the most efficient way to work in groups so there is a need within a large group to break problems into manageable pieces. An architecture describes the overall design of a system, specifies its parts and provides a reference for what is available and what still needs to be created so that developers can focus their efforts on providing value instead of reinventing infrastructure.

As humans, we all have different biases and experiences that produce a diversity of solutions. That diversity of ideas is a strength that teams use to construct novel approaches for solving problems. Unbounded, that diversity of opinions produces chaos through conflicting designs and undiscovered dependencies. An architecture provides a common vision of the system that focuses the solutions into a unified strategy. It promotes reuse and consistency by restricting the design and implementation of the system component to a known set of patterns. Those patterns allow developers to move between components and aids in the knowledge and understanding of the system. An architecture documents those patterns to remove mystery and surprise from the effort.

By definition, there is no perfect architecture. Otherwise, we would all be using it and there would be no need to discuss best practices or optimal solutions. The architecture should be the best fit for the particular problem space within the constraints of the business and available resources. Each team member may well develop the most perfect design for an individual component within the system and that design could be in complete conflict with the equally elegant and refined solution developed by her teammate. The architect must choose, from among the available solutions, to define a framework that defines the interfaces and common elements of the system. The architecture will not be perfect, but it will provide a common design philosophy that can be carried through the entire system and unite the disparate pieces into a unified whole.

3.2 Elements of Architecture

First and foremost, an architecture provides a high-level perspective of the system design. It defines the basic system strategy for interactions such as message passing, central database or client-server transactions. It describes data flow within the system, protocols for sending and receiving data, interfaces for system-to-system and user-to-system interactions and how elements will be reused or created for the system.

From there an architecture defines a framework for design and development. It defines the components of the system and their hierarchy. It lists the types of components, describes their character and relationships and specifies how they interact with others. It describes common infrastructure including common interfaces, connective components, base classes and resources.

An important consideration for the architecture is the hierarchy of authority within the system. Each resource within the system should have a single owner with authority over it. When authority is not clearly defined, multiple agents can attempt to exert authority over the same resources. When not well managed, these conflicts can produce race conditions that cause unexpected and unstable system behavior. In the classic race condition, the result is determined by which agent finishes last. Execution order may vary because of slight differences in timing. The confused software engineer may then be confronted with a system that works correctly in the debugger only to fail intermittently under normal conditions. Race conditions are notoriously difficult to diagnose and debug so the best solution is prevention. An important role of the architecture is to clearly define the resources of the system and which agent controls that resource and where that control originates.

The architecture serves as a framework for detailed design. At inception of a project, the architecture should minimally define and describe the core features of the system infrastructure with enough detail to allow in-depth design of system components. The architecture should evolve over time as new problems are discovered or expected problems become evident as product requirements are more fully defined and understood. When limitations are discovered in the architecture, the architecture should evolve and its evolution should be documented so that it continues to be relevant and provide guidance.

The architecture serves as a development roadmap allowing the development team to quickly discover what needs to be accomplished to bring a large system to life. It should be a focus of discussion to determine what the system is and how it will be constructed. It must guide the in-depth design of the system. Designs that are contrary to the architecture need to be reevaluated and revised or the architecture must evolve. One could develop without a roadmap, and some may aspire to do so, but it would be easy to get lost without one.

3.3 Modular by Nature

By its nature, an architecture is modular in since it provides a high-level description of a system by identifying its parts. A monolithic design might be reasonable for a small system but as the size of a system grows, the interactions and, therefore, connections between units grows at a rate of $n \cdot \log(n)$ which is faster than the growth in the number of its parts. The complexity of that system would quickly overwhelm the ability of people to accurately understand it. Rarely will you find a complex program that follows a single sequence of operations. There are always exceptions and deviations in a process. Minimally, a system that interacts with users must account for errors and basic human interaction. Without modularity, software quickly turns into what is affectionately and derisively known as spaghetti code.

Modularity minimizes the number of connection points and, if done well, defines their interfaces to produce well-defined APIs. Modular units may be tested in isolation with unit tests providing predictable and consistent units of functionality. A well-defined set of units with consistent behavior and spheres of influence simplifies the organization of a system making it accessible and understandable. Modularity allows people to focus their attention on discrete units within the system. A well-defined module has a discrete purpose, defined behavior and known interface.

3.4 K.I.S.S

Modularity promotes simplicity and simplicity should be a goal of every architecture. An architecture that requires complex interactions will induce, rather than reduce, errors in a system. The architecture should hide the complexity of a system in well-tested interfaces so that the complex operations can be used within the system without the need for individual developers to know the details of that complexity. Much like the libraries of a modern programming language, the infrastructure of an architecture should make it simple for developers to use the architecture so that they can focus on producing value for the customer.

In practice, I have seen many examples of complexity for the sake of complexity. It is tempting to build complex structures with several layers of nested objects and complex interactions of parts. Complexity in design is often a symptom of not truly understanding the problem or sufficiently developing a solution.

In the process of developing a solution, it is often helpful to throw down a prototype design as a brain storming exercise to start the design process. From there, the designer works with the design to discover its patterns and iteratively refines and reforms the design. Too often, inexperienced designers start coding immediately and become too vested in the implementation to adequately refine the design. They attempt to force the prototype design into a finished design by adding layers of complexity. Simplicity takes time, thought and contemplation to discover the interactions between components and reduce the design to its core. The process of design prototyping, refactoring and refinement is best accomplished on paper where the process can unfold without the being vested too early in the implementation of the design.

As a developer, when faced with a complex design, my first impulse is not to delve in and try to make sense of the design for the good of the team. My first impulse is to run away as quickly as possible to avoid the inevitable frustration of confusion and prolific cursing. In the face of complexity, the natural impulse of the sane developer is to seek a better solution that can be understood and maintained.

Simplicity should be the goal of every architect because simple is easier to understand, easier to explain, easier to use and easier to avoid mistakes. There may be complexity behind the curtain but the interfaces that the architecture provides should be simple and approachable. Simplicity is also the hallmark of elegance and for an architect, that can be quite satisfying. So, the architect should follow the edict of Keep it Simple, S….

3.5 Environment Influences Software Architecture

An architecture does not stand on its own. It is driven by needs of the customer, the needs of the developers and the needs of the business. The architecture may be the first step in the delivery of a roadmap of products that can sustain a business into the future or it may be the continuation of a product platform for continued development. It may be structured around a process of continuous delivery or by the periodic release of functional boxes. Products that are purely software will be constrained by the hardware that they run on and whether the hardware is handheld, desktop or virtual. Products that include both hardware and software will be both constrained by the hardware and the need to support the hardware through its expected life cycle and freed by the ability to define the hardware to fit the needs of the product.

There may also be regulatory requirements such as FDA certifications and compliance to industrial standards for safety critical systems that regulate not only the behavior of the product but the process of developing and testing the product. Where safety is a critical concern, the architecture must be structured to facilitate rigorous processes and development standards.

3.6 Platform Development

Often, the worst thing a business can do is build a platform. I have seen several platform efforts fail through lack of focus and vague requirements. The nature of a platform is to deliver a complete architecture but that effort makes the grand assumption that the extent and capabilities of a platform are known before any products are developed and usually before customers are consulted. The chief failure of a platform development effort is the lack of a concrete definition of doneness. A platform development effort is often the pretext for allowing the architect to go wild and throw in everything that might be needed for a platform, including the proverbial kitchen sink.

Another risky proposition for a business is to develop a platform in a central group. It makes sense to share and reuse software but a centralized platform effort may be delivered as an edict from management to reuse software developed by an isolated group attempting to serve too many stakeholders. Such efforts run the risk of producing edifices to system complexity as they attempt to address conflicting and misunderstood product requirements.

The result of a platform development effort is often a platform that does not fit the target product category. It often fails by delivering too much but not enough of what the product really needs. In the past, I have observed a platform development effort that resulted in the acquisition of expensive shelfware after the development team bought into a promise of blissful development through technologically advanced tools. In another, the platform was simply too big and too expensive to build into a product.

When given the opportunity to define and develop a new product, I and the other members of my team rejected the existing platform and decided instead to develop our own. In our calculation, the platform simply did not fit our needs and the effort to fit the platform to our needs would be wasted. The existing platform was simply too expensive and too complex. We chose a simpler path are much happier for the effort.

The architect must remind the team and the business leaders that platforms and architectures are not static creations. They grow and evolve over time. The best architectures are tailored to serve the products. The best platforms accelerate product development rather than constrain products into a particular mold. At the end of the day, the business must deliver a product to the customer. A properly designed and fitted architecture makes that effort possible.

3.7 Just in Time Platform Development

When developing what eventually became our new product platform, our mantra was “product first, but leave the door open.” That mantra expressed the view that the product development team should focus on the development and delivery of a product that the business could sell. At the time, it was a very agile philosophy before agile became popular. That focus on product allowed the development team to focus architectural development on features that were most needed by the product that used it. But, in developing that architecture, the team chose options that did not overly constrain the architecture and allowed for current and future flexibility. That flexibility allowed the architecture to evolve over time. As new features were added to the platform, the architecture evolved.

Flexibility in the platform manifested itself in several elements of the architecture. We chose a message passing architecture based on experiences with imitation in the previous architecture. We specified that message transactions were atomic to limit the interactions between modules and defined “datatags” to identify the data that was modified by those messages. We defined those datatags in “schema” files that were specifically not written in C so that the user interface could be decoupled from the core application. That decoupling allowed user interface development to proceed in parallel or in advance of development of lower-level code as long as the datatag was added to the schema file. Schema files defined the data that formed the primary interface for every module of the system to provide a consistent and universal interface.

Message types described the actions that could be performed the data identified by the datatags. Like the SNMP protocol, the initial message request types were Get and Set with response types of Response, Error, ACK and NAK. As the system evolved we added Alarm and Status messages to account for asynchronous events and by-products of other operations. The current system adds Data and Post message to limit the exposure of messages sent internally within the system and to manage message traffic.

Type	Class	Description
Get	synchronous request	A message sent to request the value of a system variable, parameter or hardware status.
Response	unicast response	A message sent in response to a successful Get request. The message payload contains the value requested by the application.
Error	unicast response	An error message in response to an unsuccessful Get request. The payload contains an errno value.
Set	synchronous request	A message sent to set a parameter or value or to initiate an action within a supplier.
ACK	broadcast response	An acknowledgement message sent in response to a successful Set message. The payload contains the actual value used by the supplier. ACK messages are broadcast to all interested clients.
NAK	unicast response	An error message sent in response to an unsuccessful SET request. The payload contains an errno value.
Post	asynchronous request	A message sent to set a parameter or value or to initiate an action without generating a response for the action.
Status	asynchronous broadcast	An asynchronous message sent to announce a new value for a system variable, parameter or hardware status. A status message reflects a change in instrument state as a side effect of a related Set operation or in response to an external event. Status messages are broadcast to all interested clients.
Data	asynchronous unicast	An asynchronous message sent to provide data within the system. One typical use for this type of message is to transfer data from hardware to an application service for further processing.
Alarm	asynchronous broadcast	An asynchronous message sent to announce an exception or fault (one-shot). The payload contains alarm parameters describing the specific circumstances of the alarm.
Alarm Start	asynchronous broadcast	An asynchronous message sent to announce the start or change in a continuous alarm condition. The payload contains alarm parameters describing the specific circumstances of the alarm.
Alarm End	asynchronous broadcast	An asynchronous message sent to announce the end of a continuous alarm condition. The payload contains alarm parameters describing the current circumstances of the alarm.

Originally, messages were defined as a C structure with a basic header and a flexible payload. The payload was a union which required the software to know how the size and number of the values in the payload. The architecture quickly added a database so that software could use accessor functions to extract values from the message payload. The current message implementation is a C++ object that encapsulates the payload within the message and removes direct access to the payload. It stores the payload type within the message instead of relying on a database so that the message object can translate itself to and from JSON and manage its own payload. The message provides a reference payload type so that arbitrarily large objects can be allocated, attached to a message, distributed throughout an application and then automatically deleted.

Nineteen years ago, that architecture defined a single application written in C and C++. Along the way, the architecture moved from a big-endian PowerPC microcontroller to a little-endian ARM embedded processor and finally to an Intel desktop processor. Today, that architecture describes a constellation of applications that are written in C++, Python and JavaScript. The decoupling that was inherent in the system enabled messages to leap between application. Formerly primitive message structures evolved to become message objects that could translate themselves to and from JSON.

The combination of focus on product development with flexibility for the future allowed that architecture to evolve and continue to meet the needs of the development team, the business and customers. The most recent product development effort had a very aggressive timeline but the business was confident that they could achieve the targets because it could rely on an established, mature body of software. Product development is hard but it would be even harder without a stable foundation on which to build.

4 The Influence of an Architect on Software Quality

Incidental to the development of the architecture, the architect affects software quality in other ways.

4.1 Requirements

The architect is often the first member of the development team to be engaged in the product definition and design of the product. While the Product Owner brings customer domain expertise, the architect brings technical expertise. At inception of a product development effort, the architect begins by interacting with the Product Owner to gather and define system requirements.

Whether they are written as stories or as formal requirements, understanding non-functional requirements is crucial to development of the system architecture and determining the tradeoffs that are inherent in the definition of an architecture. The architecture provides the first effort to define the system in a holistic manner and requires broad analysis of the system requirements. The effort to use the requirements to define the system creates a necessary push back to refine and develop the requirements. It challenges the Product Owner to more fully define the requirements and engage in discussions the further define the product and its value to users. The collaborative effort between Product Owner and architect provides an important tool to allow the process of generating requirements to stay ahead of the development effort.

It is advantageous for the architect to actually write product requirements because the architect can bring a technical rigor to the process. As a consumer of the requirements, the architect is familiar with the kind of details that developers need to implement the requirement in software. It is also a convenient way to clarify the understanding of the requirements for the architect to express his understanding of the user stories to the Product Owner through the writing of requirements.

4.2 Architectural Evaluation

As a matter of due diligence, it is incumbent on the architect to validate the architecture. For validation of efficiency and performance, that evaluation may require building a prototype to verify the performance expectations of the completed system. When done correctly, prototyping removes risk from the architecture and allows the development team to make informed decisions. However, the findings of a badly executed evaluation can have serious repercussions on the success of the architecture and product.

Our architecture once had a flirtation with Java. The performance of Java on our embedded ARM processor was a concern for the development team but the designated architect at the time had a fondness for Java. The cursory evaluation of Java performance appeared to be adequate and thus the decision to proceed was granted. However, after the system was implemented, it became quite apparent that processor was not up to the task. The performance evaluation failed to account for the prolific use of floating point operations in the Java libraries. On an embedded microprocessor that did have floating-point support in hardware, those operations had to be emulated in software. It also failed to test performance in a multithreaded environment that resulted in sluggish user interface performance and very frustrated users.

After the several attempts to salvage the situation, the product was shipped late to market. However, plans to remove Java from the platform began well before the product shipped. The flirtation with Java resulted in years of wasted man-months of effort and many lost weekends in a valiant effort to ship the product as promised despite the high hurdles imposed by an architectural decision.

4.3 Selections of Tools and Environments

Two key elements of the architecture are the tools and environment in which that architecture lives. Often that choice is dictated by the platform. Where there is a choice, the decisions of the architect can have a profound influence on the quality of the code.

For example, in embedded system, the common myth is that C++ is too big and inefficient to run within the constraints of an embedded system with its limitations of memory and CPU. That may be relevant for tiny 8-bit processors and it may have been relevant before GNU compilers became the compiler of choice for many embedded processors. However, the more rigorous type checking available in a C++ compiler allows the compiler to protect against a wide variety of errors especially in systems that often use integers of specific sizes, like embedded systems.

I once worked on a product that reused user interface code from a previous product which was written in C. However, the new product design provided a tiled display so that instead of one instance of the UI, there were essentially four instances on the screen at the same time. The developer decided to expand the original UI by replicating the parameters of each instance in an array rather than encapsulate the instances of each interface in separate objects. Instead of a neat modular construction of discrete objects the resulting monolithic spaghetti code was plagued by inconsistencies due to crosstalk between the tiles. Programs written in C tend to follow very functional constructs. While object-oriented programming is possible in C, the language does not lend much support for the effort.

As a first step in the evolution of the UI, we changed all the file name extensions from .c to .cpp so that we could compile them with the C++ compiler. After fixing compiler warnings, we then proceeded with the real task which was to re-implement the spaghetti as objects. The choice to allow C for development was pragmatic but it led to the development of a monolithic and enormously complex code. The effort to refactor the UI was a three-month hit on the schedule that was welcomed by management as a necessary exercise to stabilize the product and enable further development.

In the current iteration of the architecture, the core application has been standardized on C++11. I once described C++ as “C with object-oriented extensions.” As architect, I chose C++11 because it is the first fully object-oriented implementation of C++ because it includes the standard library as an integral part of the language. Using C++11 allowed me to implement new capabilities in the architecture to make it simpler to use.

Another important consideration for the architecture is whether to build or buy. In that sense, “buying” includes the use of open source software. However, in choosing to “buy”, the team needs to know the quality of the code. Buying code for mission critical components of the architecture has inherent risk especially if there are few alternatives. Open source software is not free but it is provided “as is” which means that the product team assumes responsibility for maintenance. Some open source also have licensing provisions that could place the entire source code at risk if used in a way that allows copy-left provisions to attach to proprietary code. Good acquisitions can significantly reduce time to market and provide proven, tested code. Bad acquisitions can increase risk and eat time and effort in debugging that could have been better spent on creating value for customers.

4.4 Documentation

The architect must document the architecture because if it isn't documented it does not exist. The architecture is the basis for all design so it must be well understood by the development team. The architect should strive to remove ambiguity and that happens best when the architecture is described in terms that can be understood, reviewed and critiqued. The architecture document should also document the tradeoffs that shaped the architecture to ensure that they don't come back to haunt the development team. A successful architecture will also be adopted for succeeding products and by other products.

My tools of choice for an architecture document are a good document editor and a graphing program. I use Word and Visio because they are available. I generally start with the Visio document because I think better in pictures. The Visio document generally provides a summary of the architecture and provides pages that are well suited for plastering cubical walls and conference rooms. The word document incorporates the images from the Visio document and provides additional details about the architecture.

Both documents are stored in the revision control system because if it isn't controlled, it will be lost. The revision control system provides a history of the evolution of the architecture and an authoritative source for the current revision of the document so that people know where to find it and won't be confused by old copies stuffed away in a filing cabinet. The document is periodically updated to reflect the current state of the architecture.

An architectural document is also a valuable tool in the process of on-boarding new team members. By offering training, the architect can set the stage for the architecture and share the architectural vision with the team.

4.5 Guardian of the Architecture

The architect defines or chooses the architecture and acts as its guardian but does not own the architecture. Some may choose to impose an architecture or perhaps the team chooses an existing architecture that they do not control. It is usually better to work collaboratively to create and refine an architecture. The architect must achieve buy-in from the development team because they are the ones who must live within its constraints.

As guardian of the architecture, the architect has a responsibility to ensure that the architecture is faithfully implemented. He or she continues to encourage team members to follow its guidance and collaborates with developers to ensure that it is implemented with integrity. The architect must address flaws in the architecture and be flexible to adapt the architecture as needed so that the architecture remains relevant and useful. After all, there is no perfect architecture and the architecture should evolve.

However, sometimes it is the role of the architect to say "no". If the architect is too flexible, the architecture may become unfocused and divergent. But, in saying "no", the architect must explain why that response is best for the architecture and not just a personal bias. Sometimes, that reason is in the service of consistency. Where necessary, the architect may provide guidance in the use of the architecture in the form of examples, coaching or training.

4.6 Quality Assurance

As part of the process of defining the architecture, the architect may also establish coding standards and design principles. Coding standards are not necessarily the same as code style. Coding standards include such standards as the MISRA standards for C and C++ which began life subset of the C language for preventing defects in automotive control software and has since been adopted by other mission critical applications.

An architect may also establish a build environment which includes a framework for automated unit tests or implement automated source code analysis that tests against known, insecure coding particles from the Common Vulnerabilities and Exposures database.

An architect should participate in design, code and test plan reviews. Design reviews allow the architect and team members to participate in the refinement of the architecture. Detailed code reviews provide a valuable tool for an architect to monitor the implementation of the architecture and to provide insight and advice on its implementation. Test Plan reviews allow the architect to ensure that system requirements have been adequately specified and provide insight for test coverage

As a senior member of the development team, the architect also can provide the benefit of a wealth of experience for other members of the team. As an active member of the team, the architect can share that experience with fellow team members.

4.7 Implementation

An architect benefits from participation in the implementation of the architecture. Through personal experience, the architect can “feel the pain” and quickly determine whether the architecture is sufficient or deficient. Personal experience can often lead the architect to proactively enhance the architecture to better fit the needs of the developers.

As a senior developer, the architect may also be the most capable person to implement key infrastructure components of the architecture. Part of that is aided by first-hand knowledge of the architectural intentions and an appreciation for how that work fits into the whole.

4.8 Qualities of an Architect

An architect is a seasoned technical leader with years of experience that provides technical insight to conceive and construct the overall design of a system. An architect must work at a detailed level without becoming lost in those details or lose sight of the larger picture. An architect must be creative to imagine novel solutions but disciplined to focus on the crucial elements of the design. An architect must organize the design in a meaningful and approachable form and communicate that vision to others. An architect must listen for concerns and requirements so that they may be addressed by the design. An architect must collaborate to incorporate the best ideas and negotiate to find the most acceptable trade-offs. An architect must be confident in advocating for the architecture but diplomatic in managing and protecting its integrity. An architect must be critical to adequately assess the strengths and weaknesses of the architecture and humble enough to continuously work to improve it.

5 Challenges for an Architect

Life, however, is not without its challenges and the business pressures may interfere with the ability of the architect to fully fulfill his obligations.

5.1 Time-to-Market Pressures

Agile processes are efficient in focusing the team to quickly adapt to market needs and drive development teams to produce more quickly and efficiently. However, that same drive to deliver quickly may also drive the development team to take shortcuts. In some cases, code intended to prototype a behavior may be deemed good enough to ship regardless of the technical debt that would be incurred by that decision.

While it is an advantage for an architect to participate in the product development, they may also be consumed by it. When an architect is enmeshed in the day-to-day process of delivering story points and contributing to the sprint velocity of the team, the needs of the sprint may override his obligations as guardian. In the Scrum process, burn up and burn down charts measure the progress of the sprint. Management may look upon those values exclusively simply because they are available and easy to measure. The role of the architect is less easy to measure because the process lacks a metric for architectural integrity.

The challenge for the architect is to strike the happy medium of being engaged in the development effort but not to be so consumed by it that he cannot fulfill his role as guardian of the architecture. The architect must also balance the need to define the architecture for the next project while the development team is feeling the pressure to finish and deliver the current project as quickly as possible.

5.2 Distributed Project Teams

Distributed project teams also present a challenge because of the lack of one-on-one interactions with team members in remote locations. The architect must be able to share his vision with those team members despite the distance or even differences in time zones. For interactions with team members in a remote location, the architect must rely more on written communications including the architecture document.

Depending on the nature of the product, the architect may also have to interact people outside the company such as contractors or even customers who must interact with the architecture through more formal definitions and interfaces.

6 Conclusion

In any large endeavor, one can choose to simply dive in and start implementation without the benefit of a road map. Certain aspect of the design will certainly reveal themselves but it is also easy to lose one's way.

An architecture provides a roadmap for development and a framework for design and implementation. It provides the high-level perspective that can guide and focus an effort. The infrastructure that an architecture provides can simplify implementation and allow developers to focus on adding value to the software. Using an evolutionary approach, an architecture can evolve with the products that it supports. The architecture may also live beyond the span of a single product not only providing the consistency within a project but across an entire platform of products.

As guardian of the architecture, the architect plays a key role in fulfilling the promise of the architecture in the product development effort. The level of integrity with which the software implements the architecture improves software quality through consistency of design and familiarity of implementation that produces software that is easier to review and easier to test. Adherence to an architecture can be a measure of quality within the system. The architect encourages a shared vision that forms the basis for development. The production of an architecture document provides a vital resource for the team that enables new team member systems to quickly gain an understanding of the system.

As a senior member of the development team, the architect can be a positive influence for quality by encouraging good design and sharing the wealth of his experiences.

References

Webb, Brian. "Programmer vs Engineer vs Architect." http://brianwebb.org/programmer-vs-engineer-vs-architect/ (accessed 6 July 2017)

Brown, Simon, 2010, "Are You a Software Architect." InfoQ. https://www.infoq.com/articles/brown-are-you-a-software-architect (accessed 6 July 2017)

Kostenko, Constantin. "Responsibilities of a Software Architect." SoftwareArchitectures.com. http://www.softwarearchitectures.com/responsibilities.html (accessed 6 July 2017)

Perrin, Chad, 2010. "The danger of complexity: More code, more bugs." Tech Republic. https://www.techrepublic.com/blog/it-security/the-danger-of-complexity-more-code-more-bugs/ (accessed 6 July 2017)

MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems. Nuneaton: MISRA Limited, 2013.

MISRA C++ 2008: Guidelines for the Use of the C++ Language in Critical Systems. Nuneaton: MISRA Limited, 2018.

Jones, Nigel, 2002 “Introduction to MISRA C”, embedded.com. http://www.embedded.com/electronics-blogs/beginner-s-corner/4023981/Introduction-to-MISRA-C (accessed 14 August 2017)