Architecture and Design Practices for Agile Project Management

by Dr. David F. Rico, PMP, ACP, CSM

A major principle within lean and agile methods is "*No Big Design Up Front* (*BDUF*)." Instead, agile teams promote the notion of evolutionary, emergent, and iterative design. Proponents of traditional methods believe a comprehensive top-down architecture and design are tantamount to system quality and project success. However, the lean and agile community has since learned that large architectures and designs are a form of "*waste*." That is, traditional teams over-specify the system architecture and design by gold-plating it with unnecessary features that undermine system quality and project success.

How do agile teams perform architecture and design? Is there such a thing? What are its associated practices? When is it performed? How much effort is applied? Try to remember some of the characteristics of agile projects. Their goal is to rapidly produce a precious few system capabilities and customer requirements that have the biggest bang for the buck! That is, produce something with high business value or great return on investment. Furthermore, system quality, project success, and customer satisfaction are ultimately achieved by significantly reducing the scope of the system architecture and design.

"*Less is more*" for agile projects! Recapping, agile projects address a smaller scope, fewer requirements, and shorter time horizons. They focus on a few critical customer needs, business functions, and capabilities. They are optimized for project success, system quality, customer satisfaction, and business value. Conversely, traditional methods are based on the theory of comprehensive, all-encompassing architectures and designs to anticipate every conceivable customer need. However, "*wisdom has proven right by her children,*" as traditional project failure rates were too high, resulting in agile methods.

Given these assumptions, the scope of an agile project is limited to a near-term release plan spanning 9, 12, or 15, months, give or take a few. Therefore, an agile architecture and design should be right-sized to fit the scope of the release plan and no more. This is true whether the architecture is created in a traditional top-down or agile bottom-up style. This means that an agile architecture and design can be visualized within the initial release planning phase when a lightweight plan is created and the most business value to a customer is achievable. Here are some of the practices for agile architecture and design.

- Prior Experience (*Prior Experience/Architecture Reference Designs/No Big Up-Front Design Necessary*): Agile, like traditional methods, assume developers have prior, pre-defined domain expertise and knowledge of architecture and design solutions. Agile developers should also have a history of past-performance, working knowledge, and experience with similar solutions (*i.e., I've already built data warehouses before, so I intuitively know what to do*). Therefore, agile methods assume that a large, up-front architecture and design phase is not necessary. That is, a chief architect or designer can simply begin implementing customer requirements (*user stories*) to realize a preconceived architecture. (*This is an important assumption. For instance, if you need to build a petabyte-scale solution, a domain expert will know that a terabyte-scale technology such as a relational database management system won't work.*)

  (***Case Study****. On a recent agile project, prior experience with technologies, architectures, and designs necessary to satisfy high-performance user stories proved invaluable. Developers were faced with creating a petabyte-scale solution. They knew typical terabyte-scale servers and technologies would not suffice. They devised a high-level architecture, purchased custom computers, and obtained middleware necessary for petabyte-scale processing. They then assembled the system, connected it to the enterprise IT infrastructure, loaded it with data, and rapidly created a prototype to demonstrate the ability to perform petabyte-scale processing. Their success came on the heels of several failed traditional projects that attempted to use terabyte-scale technologies for petabyte-scale user stories. The project was highly-successful and demonstrated the importance of finding developers with the right experience for highly-specialized requirements*.)

- Release Planning (*Visioning/Strategic Planning/Roadmapping/Capabilities Analysis*): Release planning is a lean, lightweight, and flexible form of project planning. A number of critical products are created during this stage, such as vision statements, project charters, scope statements, and high-level customer requirements. Customer needs may be in the form of enterprise-level capabilities (*i.e., very large requirements*). These are often called epics or themes. User stories may be grouped into capabilities, epics, and themes. Each release may then address one of these groupings. A release plan encompasses multiple groupings over a period of 9, 12, or 15 months (*more or less*). Multiple teams may be necessary for larger projects (*i.e., customer teams, product planning teams, and project-level teams*). Frequently communicating vision statements is a critical success factor and release planning may be used to create high-level architectures (*if necessary*).

  (***Case Study****. On a recent agile project, a vision statement, project charter, scope statement, and system metaphor was created for developers. System examples were provided to developers, along with commercial-off-the-shelf components and web services. A small list of user stories were created and prioritized related to the vision, product description, and system metaphor. The product owner described the product to be built, answered numerous questions, and urged the team to "git-r-done" above all else, i.e., spare the bureaucratic practices associated with traditional methods. Developers suggested new user stories, they were reprioritized, and a release plan was created. Developers took it upon themselves to create iteration plans, perform day-to-day self-organization, and take responsibility for implementation details. The team successfully completed the product on-time and on-budget and cited the clear vision as the most important success factor*.)

- Iteration Zero (*Early Architectural Iterations/Early Iterations to Explore Technological Alternatives*): Agile teams may establish the IT infrastructure before iterations begin. They may stand-up servers, operating systems, middleware, database services, GUIs, utilities, and other development tools in-advance. They do this in what is known as "*iteration zero*," where the first iteration is consumed standing up the IT infrastructure upon which the application is built. Agile teams may even devote two or three iterations for standing up a more complex IT infrastructure. If the system architecture proves to be a sticking point, agile teams may even devote a few iterations to establishing an architecture (*i.e., best alternatives for an anti-lock braking system, avionics architecture, network infrastructure, data warehouse, petabyte-scale solution, etc*.). The first few iterations may be used to explore technological alternatives (*i.e., which commercial web services to use*?).

  (***Case Study***. *On one agile project, developers had the technical skills to stand up an IT infrastructure. A development environment, tools, and technologies were suggested by the customer. The developers stood up servers, operating systems, web servers, middleware, databases, and development tools. They also procured Wikis, agile project management tools, version control software, and other collaboration tools. Much of this activity was performed during the release planning phase. However, the customer allowed an "iteration zero" as a means of finishing the IT infrastructure. The team rapidly prototyped a proof-of-concept to verify the IT infrastructure worked. Although this was a successful project, the team felt the customer should have stood up the IT infrastructure to minimize the intense activity prior to and during "iteration zero." Common IT infrastructure is an oft-cited success factor, especially for distributed teams such as this one*.)

- Emergent Design (*User-Story-to-User Story Design/Design Patterns/Refactoring/Generalizing*): Emergent designs evolve "*user-story-to-user-story*" and "*iteration-to-iteration*." A small set of prioritized user stories are implemented regardless of dependencies between them. Developers do not logically order user stories or iterations, or make other architectural assumptions. Emergent design has proven to work in many cases. User stories describe customer needs that encompass a vertical slice of functionality from the GUI to the database (*i.e., I want to order a book*). Therefore, agile developers create a GUI, middleware, database queries, database schema, and infrastructure capabilities for each user story in isolation (*i.e., autonomous user stories*). A barely-sufficient architecture is thus created, free from waste, gold-plating, and unnecessary functions. This reduces costs, improves quality, and maximizes customer satisfaction and business value.

  (***Case Study***. *A web developer was asked to build a non-enterprise, departmental-level system. His customer did not know what the requirements were in-advance. His first action was to create a multi-year project schedule, which included long requirements, architecture, and design phases. Instead, an agile coach advised him to build a rapid prototype of the system based on his best assumptions and demonstrate it to his customer once a week. His customer began feeding him requirements after each demo. The developer finished the system within 90 days to his customer's satisfaction and began demonstrating it at the enterprise level. His customer was promoted and given a raise for the brilliant new system. The developer also received a substantial cash award for his efforts. He had no prior knowledge of agile methods. He knew how to stand-up an IT infrastructure to support his application, which was a critical success factor*.)

- Just-Enough Architecture (*Lean/Lightweight/Just-in-Time/Lean and Agile Architecture and Design*): Just-enough architecture may be needed based on the complexity, scale, or scope of the system. A rough blueprint of enterprise or system architecture is created instead of using emergent design (*i.e., high-bandwidth communication lines, extraction-transformation-and-load functions, database schema optimized for volume and performance, middleware for interfacing to applications, business objects for data mining and reporting, GUIs for end-user operations, etc*.). However, this shouldn't be months, years, or decades (*i.e., maybe a few hours, days, or weeks*). The goal is to establish an architecture, design, or technological vision and platform for moving forward that addresses a near-term customer capability. This could be done during release planning, appear as an explicit architecture phase, or be performed in a few early iterations.

  (***Case Study***. *Database developers may refuse to evolve schemas in an agile and iterative fashion. They want to identify all customer requirements in-advance. Then a large schema is developed over a long period of time. Once the perfect schema is done, it will never be changed. Their goal is to establish enterprise-level schemas vs. individual departmental databases. However, enterprise databases can be developed in an emergent, iterative, day-to-day style. It is possible to create a just-enough, just-in-time, lean, lightweight, and agile enterprise schema for enterprise purposes that evolves from iteration to iteration. Enterprise repositories may evolve in less frequent cycles, while departmental databases evolve more frequently. Just-enough enterprise schemas yielding near-term business value are both feasible and desirable in today's environment vs. big, upfront, multi-year, multi-million dollar schemas containing waste to anticipate low-priority customer needs*.)

- Product Owner Involvement (*Product Owner/Customer/Coach/Mentor/Scrum Master Collaboration*): Product owner involvement is a critical success factor. They must create project visions, charters, and scope statements. They develop prioritized user stories based on business value, risk, and other factors. They frequently visit teams, reinforce vision statements, and answer questions on an as-needed basis. They may serve as agile project managers who guide teams with a light touch and heavy doses of emotional intelligence (*i.e., people skills*). Some view them as a single-wringable-neck, but the better term is "*project champion*." They must proactively monitor project performance, get the resources teams need, stave-off ornery stakeholders, shield teams from politics, clarify user stories, review designs, and evaluate demos. They institute other practices to ensure success and may serve as the chief architect to guide system design (*when necessary*).

(***Case Study***. *On one agile project, developers spent about one-third to one-half of iterations clarifying requirements and discussing design alternatives. Some illustrated wire frames, screen shots, and other rapid prototypes and spikes. This activity preceded detailed design, implementation, testing, and demonstrations. Early design involvement worked better than arguing with developers to overturn design decisions made without earlier interactions, i.e., they were more willing to make design changes than implementation changes. Two system demos were instituted for iterations. One was for the product owner and the other was for the customer to ensure demos went off without a hitch. The customer was hyper-sensitive and would have terminated the project if the demo would have failed to satisfy their user stories. The developer's success was the product owner's success, and likewise, developer failure was the product owner's failure*.)

What's the bottom line? Agile methods have just-enough, just-in-time, emergent architecture and design practices for successfully creating products that satisfy customers and maximize business value. Emergent design minimizes effort, cost, waste, defects, schedules, poor morale, project failure, and customer dissatisfaction. It is the antithesis of the traditional all-encompassing, top-down "*big design up front* (*BDUF*)" to prematurely anticipate unarticulated customer needs. Big upfront architecture and design is done to lower costs and increase system quality, although it has proven to have the opposite effect.

Emergent design is an efficient and waste-free practice for creating today's complex systems. However, it is important to note that its success rests on other interrelated practices. Some of these include prior experience, release planning, and iteration zero. Oftentimes, developers have inadequate experience, are unfamiliar with release planning, or don't realize the value of standing up an operational IT infrastructure. Sometimes, project leaders do not understand the importance of release planning, creating and communicating crystal-clear vision statements, or the critical role they play in day-to-day design.

A few books that come to mind include "*Planning Extreme Programming*" by Kent Beck, "*Agile Estimating and Planning*" by Mike Cohn, and "*Agile Project Management*" by Jim Highsmith. Kent's book gives a 40,000 foot overview, Mike's book provides step-by-step guidance, and Jim's book provides an overarching framework for adapting release planning to larger and more complex projects. Although, there are a few textbooks available on emergent design, just-enough architecture, and even lean architecture, the principles of "*evolutionary design*" as it applies to agile methods have yet to be fully articulated.

In spite of the promise of emergent design as it applies to agile methods, customers, project leaders, developers, and other critical project stakeholders are not perfect. Essential non-functional requirements often go unidentified and undocumented. For instance, security, usability, performance, maintainability, scalability, reliability, availability, safety, and other critical system characteristics often go undefined. Security requirements are essential for today's petabyte-scale systems. Usability is at an all-time low and user-experience design examines the ecosystem of useful services surrounding individual computers.

Customers and developers are so consumed by trying to determine whether a system "*can*" be built successfully. Conversely, they rarely ask themselves whether they "*should*" develop the system. Just because we have the ability to store billions of photographs, credit card numbers, personally-identifiable information records, and other highly-sensitive data doesn't mean we should. The responsibility of creating complex, high-risk information processing systems has never been greater. Unreliable systems may anger customers, unsecure systems could cost billions of dollars, and unsafe systems can cost lives.

Agile project management and its practice of emergent design are a powerful combination that enables developers and customers to successfully build and acquire complex systems that create business value for their enterprises. In the early days, traditional developers believed agile methods sped up development (*i.e., shortened cycle time*) at the expense of design, quality, and long-term maintainability. Developers are now entering an era in the adoption lifecycle of agile methods where we have the evidence to show that we can achieve superior designs, ironclad quality, and lower overall total lifecycle costs.

Dr. Rico has been a leader in support of major U.S. gov't agencies for 25 years. He's led many Cloud, Lean, Agile, SOA, Web Services, Six Sigma, FOSS, ISO 9001, CMMI, and SW-CMM projects. He specializes in IT investment analysis, portfolio valuation, and organizational change. He's been an international keynote speaker, presented at leading conferences, written six textbooks, and published numerous articles. He's also a frequent PMI, APLN, INCOSE, SPIN, and conference speaker (http://davidfrico.com).