Lessons in Agility from Internet-Based Development

Scott W. Ambler, *Ronin International*

The author describes two Internet startup companies that adopted effective and efficient modeling and documentation practices, one taking a communal, team-based approach and the other a chief-architect approach. They found a "sweet spot" where modeling efforts can provide significant benefit without incurring the costs of onerous documentation.

At the peak of the Internet revolution hype, we were inundated by claims that the fundamental rules of business had changed. To support this new business paradigm, new companies sprang up virtually overnight, often delivering very good software very quickly. I wondered whether the rules of software development had also changed. Were we witnessing a paradigm shift in the way we develop software? Was there something to the concept of "development in Internet time"?

In April 2000, the Internet bubble burst and the business world was brought back to reality, discovering that business fundamentals hadn't changed. However, it isn't so clear whether the fundamental rules of software development have changed.

I worked as a software process consultant to two Internet startups during the boom—let's call them XYZ.net and PQR.com to protect their identities, as both are still in business. XYZ.net focused on fundamental infrastructure services for Internet users and, when I joined them, was about to start work on its third generation of software to support its upcoming initial public offering. PQR.com was an online electronic retailer in the process of re-architecting its system to support its projected growth and to position itself for a future IPO. The two common factors between the companies were my involvement and the fact that both organizations were facing phenomenal business growth and thus needed to re-architect and redevelop their systems. They were also hiring new developers to help them deliver this new software: XYZ.net had grown from three to 30 developers in less than a year, and PQR.com had grown from the original two founders to 25 developers in a similar time frame. Both companies felt they needed a more mature software process that included software modeling. Both organizations had very young staff—the average age was in the mid-20s—and both had team-oriented cultures.

Furthermore, both companies wanted to define, and then train their staff in, a version of the Rational Unified Process[1] tailored to meet their specific situations, with extensions from other software processes.[2,3] Both organizations wanted to be able to claim to potential investors that they were using an accredited software process yet didn't want unnecessary bureaucracy to slow them down. Both organizations needed to improve their modeling practices; senior management felt that although they had bright, talented developers, they weren't as effective as they could be. The developers at the two organizations had mixed feelings about this, with attitudes ranging from "it's about time" to "modeling is a waste of time; all I need is source code."

The tailored version of the RUP needed to include an enterprise architecture modeling process that would define their technical and application infrastructures. This enterprise architecture effort was beyond the RUP's scope, which focused on the software process for a single project instead of a portfolio of projects. Furthermore, both organizations needed effective approaches to their modeling and documentation on individual projects. Although the RUP clearly included sophisticated modeling processes, each organization felt the RUP approach to modeling was too cumbersome for Internet-based development and wanted to find a better way to work.

Communal architecture

To architect the new generation of XYZ.net's software, the company introduced an architecture team comprising the technical leads from each development team, the vice president of systems, the chief technology officer, the quality assurance and testing manager, and several senior developers. The first week the team met for several afternoons to identify where they were going, what their fundamental requirements were, and then to propose an initial architecture to support these needs. When they weren't modeling the new architecture, everyone worked on their normal day-to-day jobs. XYZ.net was a round-the-clock operation: no one could simply stop evolving the existing software. This had the side benefit of giving people time to reflect on the proposed architecture as well as to discuss ideas with co-workers.

During the architecture modeling sessions, they drew diagrams on whiteboards. At first, the team tried to use a CASE tool connected to a projector but found this to be insufficient—the person operating the tool could not keep up with the conversation, and the tool wasn't flexible enough to meet our needs (they needed more than the Unified Modeling Language[4] diagrams that it supported). The whiteboard also required little training to work with and allowed them to swiftly switch back and forth between modelers. As the team modeled, people would provide input and insights, question assumptions, share past experiences, and even talk about potential implementation strategies. The simpler tool enabled them to work together as a team. Whenever they needed permanent documentation, they would simply draw the models on paper and then scan them to produce electronic copies. Occasionally the team would capture the model in a CASE tool, but the scanner approach was more common because few developers had access to the CASE tool due to platform incompatibilities.

The team drew UML diagrams but found they needed a much wider range of techniques—some of the software was written in procedural languages, so they needed nonobject diagrams such as structure charts. They also drew dataflow diagrams to depict workflow issues and free-form diagrams to depict the system's technical architecture.

Once the initial architecture model was in place, they continued to meet almost weekly (only when they needed to). The meetings were typically one or two hours long and always included an informal agenda that identified one or more issues to be addressed. The owner of each issue was responsible for explaining it, what he or she saw as the likely solution, and potential repercussions of this approach. Often, someone would come to an architecture session with the issue mostly solved; the group would simply review it and provide appropriate feedback. Presentations were typically done in a "chalk talk" manner, in which the owner would describe an issue by drawing on the whiteboard. This worked well because many people would already be familiar with the issue, having been involved in earlier discussions with the owner outside of the architecture modeling sessions.

Introducing a chief architect

Taking a different approach, PQR.com decided to hire an experienced senior developer as a chief architect, an "old guy" in his early 30s who had built scalable systems before. He spent his initial week with the company learning about the business and the current status of the systems, mostly through discussions with senior IT and operations staff. He then ran small modeling

Both organizations needed effective approaches to their modeling and documentation on individual projects.

The main difference in their approaches was how architectural documentation was updated over time.

sessions to identify architectural requirements as well as potential architectural solutions. His approach was to hold one-hour "chalk talk" sessions involving two or three other people, followed by several hours of modeling privately with a diagramming tool and a word processor to explore and document what he had learned. He then posted his models as HTML pages on the internal network where everyone, developers and business people alike, could view them and potentially provide feedback. It was quite common for people to schedule an impromptu meeting with him to discuss his work or to simply send him an email.

Documenting the architecture

Both organizations chose to document their architecture to make it available to all their software developers. Their approaches had several interesting similarities:

- Official documentation was HTML based. Both companies chose HTML (and JPEG or GIF pictures) to support the wide range of operating systems the developers used, thus making the documentation accessible to everyone. Furthermore, the developers were intimately familiar with HTML and had existing tools to view and edit the documents.
- They created only overview documentation. To remain effective, both companies documented only overview information pertaining to the architecture, recording the critical information that developers needed to know to understand how it worked. The documentation was primarily focused around an architecture diagram showing major software services deployed within the technical environment, with overview descriptions for each software service or system and appropriate links to the documentation pages maintained by individual project teams. A printout of the architectural documentation would have been no more than 15 pages.
- The information architecture wasn't special. The databases and file storage systems appeared on the system architecture diagram along with applications and software services. The data team was treated the same as any other project team: all the teams provided functionality that other teams needed, and each team in effect was the customer of several other teams. This required everyone to work together to resolve any configuration and release management issues. This was unlike many organizations where the data team is considered a corporate infrastructure team that supports application development teams. At both companies, every team was an infrastructure team.
- Modeling and documentation were separate efforts. Both companies realized that the act of modeling and the act of writing documentation are two distinct efforts. In the past, most developers did not have significant experience modeling, so they equated it to using sophisticated CASE tools that generated detailed documentation. After experiencing several modeling sessions using simple tools such as whiteboards, after which someone wrote up the appropriate summary documentation (if any), they learned they could achieve the benefits of modeling without incurring the costs of writing and maintaining cumbersome documentation.

The main difference in their approaches was how architectural documentation was updated over time. PQR.com had a designated owner of the documentation, the chief architect, whereas XYZ.net took a more communal approach and allowed anyone to update the documentation. At PQR.com, people were required to first discuss their suggestions with the chief architect, who then acted accordingly. At XYZ.net, people would first talk with the perceived owner of a part of the documentation, typically the person initially assigned to that aspect of the architecture, and then update it accordingly. No one seemed to abuse this privilege by changing the architectural documentation to further their own political goals. One reason was that the XYZ.net developers were a closely knit team. Second, because all HTML pages were managed under their version control system, there was a record of who changed each page.

Comparing the two modeling approaches

Let's compare the two approaches to enterprise architectural modeling with regard to several critical factors:

- *Calendar time.* XYZ.net's architecture team required less calendar time to develop their initial architecture and then to evolve it over time, because several people worked on it in parallel.
- *Acceptance.* The communal approach taken by XYZ.net clearly resulted in quicker acceptance by the developers for two reasons. First, representatives of each development team participated in the architectural team, and they effectively "sold" the architecture to their teammates, often simply by asking for their help when they were working on portions of the architecture. Second, everyone could update the architecture documentation as they required, thus promoting a sense of ownership.
- *Cost.* PQR.com's approach required less overall effort because the chief architect performed most of the architectural modeling, enabling his co-workers to focus on other tasks.
- *Control.* The chief architect at PQR.com provided a single source of control, albeit one that became a bottleneck at times. The team approach provided several sets of eyes working on any given aspect of the architecture, although its shared ownership of documentation provided opportunities for mischief (although to my knowledge none ever occurred).
- *Correctness.* Both approaches resulted in scalable architectures that met the needs of their respective organizations.
- *Process compatibility.* Both approaches worked well within a RUP environment, although the chief-architect approach seemed to be better attuned to the RUP than the communal-architecture approach.

Project-specific modeling

The project process was similar at both organizations: systems were developed in a highly iterative manner and released incrementally in short cycles ranging from six to 12 weeks, depending on the project. The marketing department wrote the initial requirements documents, typically high-level needs statements. The IT department heads and the project leads did a reality check on these documents to rework unrealistic ideas before presenting them to the teams. Modeling sessions occurred as each project team required, with some team members meeting with appropriate project stakeholders to work through an issue in detail. These modeling efforts were similar to those performed by the XYZ.net architecture team: people met and worked on the models as a team around a whiteboard. Often, they would develop several models at once—perhaps several use cases along with some screen sketches, some sequence diagrams, and a class diagram. They did not have "use case modeling sessions" or "class modeling sessions"—just modeling sessions.

At XYZ.net, the development teams had one to eight members, each of whom focused on one part of the overall infrastructure. At any given time, each team would be at a different place in the project life cycle. Sometimes two or more teams would need to schedule their releases simultaneously to enable various subsystem dependencies. The individual teams knew of those dependencies because they worked closely with the developers from other teams, often walking across the floor or up the stairs to sit down and talk with them.

These modeling sessions differed from traditional ones. First, they iteratively did the minimal amount required that let them start coding, with most modeling sessions lasting from 10 minutes to an hour. Second, each modeling session's goal was to explore one or more issues to determine a strategy for moving forward; the goal wasn't simply to produce an artifact. Their goal was to *model*, not to *produce* models.

Developers worked with technical writers to produce the release notes, installation procedures, and user documentation. The release notes were the formal documentation for the subsystems, written as HTML pages and posted online so that all internal developers could see them. The release notes typically summarized the newly implemented requirements, the design (diagrams were typically scans of their primary sketches and were posted as JPEG files), and a listing of the files making up the subsystem. Luckily, XYZ.net developers were already producing release notes following this process in their premodeling days, so the only change they needed to implement was to add the design diagrams that presented an overview of their work. At first, some developers included de-

The project process was similar at both organizations: systems were developed in a highly iterative manner.

An interesting similarity between the two companies was their approach to data.

tailed diagrams in their release notes, but they quickly found that they provided little value. When the developers wanted detailed information, they went straight to the source code, which they knew was up to date.

Developers also worked closely with test engineers. The developers needed help unit-testing their software, and the test engineers needed help learning the software to develop acceptance tests for it. Before software could be put into production, the quality assurance department first had to test it to ensure that it fulfilled its requirements, it was adequately documented, and it integrated with the other subsystems.

The development process worked similarly at PQR.com, with teams of three to six people. Developers from the various teams worked closely together, helping one another where necessary. PQR.com developers took a more formalized approach to documentation, writing requirements documents, design documents, and technical release notes. The requirements documents were based on the initial marketing requirements documents and the requirements models produced from them. The requirements document was written in parallel with development of the design document and source code and was typically finalized during the last week of an iteration: the teams worked iteratively, not serially, so they could react to new and changing requirements as needed. A requirements document included a use case diagram and detailed use cases for the release as well as related business rules, technical (nonfunctional) requirements, and constraints as required. Design documents included screen specifications when applicable (typically just a JPEG picture of an HTML page) and applicable overview diagrams; also, the developers preferred source code to detailed design diagrams.

An interesting similarity between the two companies was their approach to data: their data efforts were treated just like development projects. However, the data teams worked differently than the other project teams, investing more time in detailed modeling. At both companies, the data groups used a CASE tool to create physical data models of their relational database schemas. The tool generated and reverse-engineered data definition language (DDL) code, including database triggers, thereby making the additional detailed modeling effort worthwhile.

Dissension within the ranks

Several individuals at XYZ.net didn't appreciate the need for an increased effort in modeling and documentation. This was due to one or more factors: they perceived themselves as hard-core programmers, they preferred to work alone or with others of similar mindsets, or their people skills needed improvement. Moreover, they typically didn't have good modeling and documentation skills, nor did most of their co-workers. For all but one person, a combination of peer pressure and the ability to see "which way the winds were blowing" motivated them to go along with adopting a tailored version of the RUP, albeit with a little grumbling. The lone holdout required minor disciplinary action: the head of development had a few personal chats with him, he wasn't allowed to move on from his current project until he had written the documentation and the quality assurance department had accepted it, and his promotion to team lead was put on hold for a few months.

There were few problems at PQR.com, likely because its staff was more aware of modeling's benefits to begin with. People had various levels of modeling skills, so they received training, but nobody was actively antagonistic toward increased levels of modeling as long as they received training.

Comparison with traditional project development

How was this different from traditional development practices, particularly of typical RUP instantiations? From the 50,000-foot level, it wasn't different at all. Both companies identified requirements and then architected, designed, coded, tested, and deployed their software. However, at ground level, there were some interesting differences.

First, development started with the enterprise architecture model. To fit in with and exploit other systems, all project teams started with the existing enterprise architecture model because it showed how their efforts fit into the overall whole. When the project teams delved into the details, they sometimes discovered the need to update the architecture, which they did following their organization's chosen practices, described earlier.

Development was also highly iterative. Project teams did a little modeling, coding, and testing and then repeated as necessary.

They did not take the serial approach of completely developing and accepting the requirements model, then completely developing and accepting an analysis model, and so on. This let them react swiftly to changes in their highly competitive markets and rapidly deliver software to their end users.

Also, modeling was streamlined. They modeled just enough and then got on with implementation, avoiding analysis paralysis while at the same time showing that their models actually worked (or didn't) by writing code based on them.

Moreover, both organizations used simple modeling tools. With the exception of the two data teams, the projects preferred simple sketches on whiteboards over diagrams created using sophisticated CASE tools. This sped up the modeling process, although it might have slowed development when developers could not automatically generate code from their models—a feature of many modern tools—or reverse-engineer existing code to create diagrams to help them understand it.

Finally, documentation in the two projects was minimal. Both companies improved the quality of their system documentation and managed to find the "sweet spot" where it was just barely enough to meet their needs and no more. Many traditional projects produce far more documentation than they need, reducing their ability to implement new features and sometimes even causing the project to fail completely.[5]

Table 1

The principles of agile modeling

Core principles	Supplementary principles
Assume simplicity	Content is more important than representation
Embrace change	Everyone can learn from everyone else
Incremental change	Know your models
Model with a purpose	Know your tools
Multiple models	Local adaptation
Quality work	Maximize stakeholder investment
Rapid feedback	Open and honest communication
Software is your primary goal	Work with people's instincts
Enabling the next effort is your secondary goal	
Travel light	

Table 2

Agile modeling practices

Core practices	Supplementary practices
Active stakeholder participation	Update only when it hurts
Apply the right artifacts	Use the simplest tools
Collective ownership	Apply modeling standards
Consider testability	Apply patterns gently
Create several models in parallel	Discard temporary models
Create simple content	Formalize contract models
Depict models simply	Model to communicate
Display models publicly	Model to understand
Iterate to another artifact	Reuse existing resources
Model in small increments	
Model with others	
Prove it with code	

Setting the foundation for agile modeling

My experiences on these two projects provided significant insights for what was to become the Agile Modeling methodology.[6] AM is a practice-based methodology for effective modeling and documentation of software-based systems. Principles and values guide the AM practices, intended for daily application by software professionals. AM is not hard and fast, nor is it a prescriptive process—in other words, it does not define detailed procedures for how to create a given type of model. Instead, it provides advice for how to be effective as a modeler.

This is different from modeling methodologies such as Iconix[7] or Catalysis[8] that suggest a specific collection of modeling artifacts and describe how to create those artifacts. Where Iconix promotes applying use case models, robustness diagrams, UML sequence diagrams, and UML class diagrams, AM suggests that you apply multiple models on your projects and the right artifacts for the situation. AM is complementary to most traditional modeling methodologies—it does not replace Iconix and Catalysis but instead provides complementary advice for improving your effectiveness with them. AM's scope is simply modeling and documentation. It is meant to be used in conjunction with other agile software processes[9,10] such as Extreme Programming,[11] Scrum,[12] and the Dynamic System Development Method,[13] as well as "near-agile" instantiations of the Unified Process.

AM adopts the values of XP—simplicity, feedback, communication, and courage—and extends it with humility. These values are used to derive a collection of principles (see Table 1) that in turn drive AM's practices (see Table 2). Nothing about it is truly new. It is a reformulation of concepts that many software professionals have been following for years.

XYZ.net and PQR.com were successful because they followed a collection of commonsense approaches.

Lessons learned

XYZ.net and PQR.com were successful because they followed a collection of commonsense approaches. Let's explore the lessons learned at XYZ.net and PQR.com and see how they relate to AM.

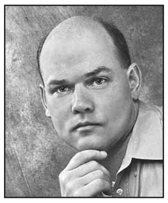
- *People matter.* Everyone worked together to make the effort succeed—it was as simple as that. We operated under a philosophy along the lines of the Agile Alliance's "value of individuals and interactions over processes and tools"[9] and Alan M. Davis's 131st principle, "People are the key to success."[14]
- *You don't need nearly as many documents as you think*. Both companies built mission-critical software very quickly, software that is still in operation several years later and that has been modified significantly since—and we did so without creating mounds of documentation. Fred Brooks provides similar advice with what he calls the documentary hypothesis: "Amid a wash of paper, a small number of documents become the critical pivots around which every project's management evolves."[15] In short, the teams followed the principle "Travel light," producing just enough documentation and updating it only when they needed to, much along the lines of AM's practice "Update only when it hurts."
- *Communication is critical*. The amount of required documentation was reduced because both organizations had a high-communication environment. Developers worked closely together, face-to-face with their co-workers and customers (either marketing and sales staff or the chief technology officer). Models were left on the walls so that everyone could see them, as AM suggests with its practice "Display models publicly." By doing this, developers could refer to the models when explaining their strategies to others and often received valuable feedback, often simply in the form of questions, from people who had taken the time to look over the models. We had rediscovered Davis's 136th principle, "Communication skills are essential" and his 8th principle "Communicate with customers/users."[14]
- *Modeling tools aren't nearly as useful as you think*. Although we honestly tried to use the leading UML modeling tool at the time, as well as an open source software offering, the tools simply didn't offer a lot of value. We weren't interested in creating a lot of documentation, and the tools didn't generate code for our deployment environment. The only useful CASE tools were the data-modeling tools from which diagrams were printed and DDL code generated. We did, however, take advantage of some simple tools, whiteboards and flip charts, and were very effective doing so. This experience reflects the findings and techniques of the user-centered design community in the late 1980s[16] and is captured in AM's practice "Use the simplest tools."
- *You need a wide variety of modeling techniques in your intellectual toolkit.* The techniques of the UML, although useful, were not sufficient—both companies needed to perform process, user interface, and data modeling. Furthermore, each developer understood some but not all of the modeling techniques that we were using, resulting in the teams stumbling at times because someone would try to model something using an ill-suited technique. These concepts are captured in AM's principle "Multiple models" and its practice "Apply the right artifact(s)"—advice similar to what everyone's grandfather has likely said at some point, "Use the right tool for the job." The concept of requiring multiple modeling techniques is nothing new; in the 1970s, structured methods such as those promoted by Chris Gane and Trish Sarson[17] supported this concept. As the teams modeled, they would often work on two or three diagrams in parallel, each of which offered a different view on what they were discussing, and would move back and forth between the diagrams as appropriate, similar to AM's practices "Create several models in parallel" and "Iterate to another artifact," respectively.
- *Big up-front design isn't required*. Although XYZ.net spent a few afternoons formulating their initial enterprise architectural strategy, they quickly got

down to work. Similarly, the chief architect at PQR.com also created an initial high-level model. Then he started to work with individual teams to implement portions of it, updating his model as required. Both companies discovered that they didn't need weeks or months of detailed modeling and documentation to get it right, that hours were sufficient. Instead, they worked along the lines of AM's practice "Model in small increments," modeling the architecture a little and then either exploring strategies by proving them with code or simply starting work on the actual software itself. In his essay "Passing the Word,"[15] Brooks provides similar advice—that architects must be able to show an implementation of their ideas, that they must be able to prove that their architecture works in practice.

- *Reuse the wheel, don't reinvent it.* At XYZ.net, we took advantage of open source software whenever we could, building mission-critical systems quickly and effectively as a result. Through open source we had rediscovered Davis's 84th principle: "You can reuse without a big investment." We reused more than just open source software; we also adopted industry standards wherever possible, existing architectural strategies wherever possible, and applied design patterns when applicable. We discovered there was a wealth of material available to us when we chose to reuse it, supporting AM's practice "Reuse existing resources."

Agile approaches to software development worked for these two Internet-based companies. Both XYZ.net and PQR.com applied existing, proven principles of software development in new ways to deliver complex, mission-critical software in a highly iterative and incremental manner. The lessons they learned apply beyond Internet development: most organizations can benefit from a shared architecture across projects as well as more effective and efficient modeling and documentation practices. Many of the "modern day" principles and practices of leading-edge development processes are the same (or slightly modified) principles and practices of yesteryear. Although the fundamentals are still the fundamentals, the way in which the fundamentals are applied has changed—agilely, not prescriptively.

About the Author



Scott W. Ambler is president of and a principal consultant for Ronin International, Inc. (www.ronin-intl.com) and author of several software development books, including *The Object Primer* 2/e (Cambridge Univ. Press, 2001) and *Agile Modeling* (John Wiley & Sons, 2002). His professional interests focus on identifying and exploring techniques to improve software developers' effectiveness. He received a masters in information science and a BSc in computer science, both from the University of Toronto. He is a member of the IEEE and ACM. Contact him at scott.ambler@ronin-intl.com.

References

1. P. Kruchten, *The Rational Unified Process: An Introduction*, 2nd ed., Addison Wesley Longman, Reading, Mass., 2000.
2. S.W. Ambler, "Enterprise Unified Process," 2001, www.ronin-intl.com/publications/unifiedProcess.htm.
3. S.W. Ambler, *Process Patterns—Building Large-Scale Systems Using Object Technology*, Cambridge Univ. Press, New York, 1998.
4. Object Management Group, "The Unified Modeling Language (UML), Version 1.4," 2001, www.omg.org/technology/documents/formal/uml.htm.
5. J.A. Highsmith III, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House, New York, 2000.
6. S.W. Ambler, *Agile Modeling*, John Wiley & Sons, New York, 2002.
7. D. Rosenberg and K. Scott, *Use Case Driven Object Modeling with UML: A Practical Approach*, Addison Wesley Longman, Reading, Mass., 1999.
8. D.F. D'Souza and A.C. Wills, *Objects, Components, and Frameworks with UML: The Catalysis Approach*, Addison Wesley Longman, Reading, Mass., 1999.
9. K. Beck et al., "Manifesto for Agile Software Development," 2001, www.agilealliance.org.
10. A. Cockburn, *Agile Software Development*, Addison Wesley Longman, Reading, Mass., 2002.
11. K. Beck, *Extreme Programming Explained—Embrace Change*, Addison Wesley Longman, Reading, Mass., 2000.
12. M. Beedle and K. Schwaber, *Agile Software Development with SCRUM*, Prentice Hall, Upper Saddle River, N.J., 2001.
13. J. Stapleton, *DSDM, Dynamic Systems Development Method: The Method in Practice*, Addison-Wesley, Reading, Mass., 1997.
14. A.M. Davis, *201 Principles of Software Development*, McGraw Hill, New York, 1995.
15. F.P. Brooks, *The Mythical Man Month: Essays on Software Engineering, Anniversary Edition*, 2nd ed., Addison-Wesley, Reading, Mass., 1995.
16. J. Greenbaum and M. Kyng, eds., *Design at Work: Cooperative Design of Computer Systems*, Lawrence Erlbaum Assoc., Hillsdale, N.J., 1991.
17. C. Gane and T. Sarson, *Structured Systems Analysis: Tools and Techniques*, Prentice Hall, Upper Saddle River, N.J., 1979.