

A resounding Yes to agile processes -- but also to more

Ivar Jacobson

Vice President Process Strategy

Rational Software Corporation

Agility has become today's buzzword when describing a modern software process. Everyone is agile. An agile team is a nimble team able to *appropriately* respond to changes. Change is what software development is very much about. Changes in the software being built, changes to the team members, changes because of new technology, changes of all kinds that may have an impact on the product they build or the project that creates the product. Support for changes should be built-in everything we do in software, something we embrace because it is the heart and soul of software. An agile team recognizes that software is developed by individuals working in teams and that the skills of these people, their ability to collaborate is at the core for the success of the project.

For purpose of illustration here is an agile project as described by a project manager¹.

“I took over the job as project manager Nov 2000 at which time the team consisted of four people. They had developed a simple prototype barely demonstrating the feasibility of the idea: using intelligent agents to support the individual developers in software development. The product would include a knowledge base described as a large number of rules, formulated in a rule language, a compiler generating executable code from the rule language and a rule engine – not exactly something I had done before. The company was a startup company funded with seed money only. We needed to get a first workable product through the doors as quickly as possible, be extremely focused.

In the first phase we developed a vision in writing for the first product release. We needed as a team to know where we wanted to go. We also needed to get consensus on this vision with sales so that we would get something we believed we could sell. We were in particular explicit on what we wouldn't do. We knew that the released product would not be all we said, it would contain something less and maybe something more. This took about a month and I grew the team to seven people, four of the team developed the compiler, the rule engine and generic GUI and three of them developed the knowledge base. The team was very competent in their respective roles.

In the second phase we developed a skeleton, executable system from which we could grow to the first product. We worked out the software architecture. We decided on software platform (C# and .NET). We described each use case and made a simple object model using UML (we knew Rose so we used that). We worked with mock-ups and story-boarding to design the user interface -- the team had now grown with a creative and experienced UI designer. We also identified all significant risks and worked through them all. To verify our work we implemented and were able to run the use cases that mitigated the risks. This took about three months. During which time we iterated the solution twice. There was some significant re-work during this period based on customer

¹ Svante Lidman, svante.lidman@jaczone.com, at Jaczone AB, a company in which I am engaged on its advisory board.

interactions and for technical reasons. That was no surprise to me. We had to adapt our solution as we better understood what we were doing before we thought we had something we could grow from. At the end of this phase we felt comfortable in committing to a release date. We still didn't know exactly what we would deliver but we knew it much better than after the first phase.

The third phase was basically growing the product in several internal releases. Thanks to having spent so much effort on the skeleton system in the second phase, we could grow the product quite smoothly. That doesn't mean that we didn't have to do some changes, but no major ones. We could move forward as we had committed. We didn't do all we said we would do at the end of phase two but nothing we cut off was very important. Since I was absolutely committed to release a "defect-free" product for our first customers we spent the necessary time on peer reviewing and testing of each internal release.

In the fourth phase we released the product to a handful of customers committed through contracts to work with the product. During this phase we didn't expect to find many bugs – and we didn't. We wanted to know if the excitement that customers felt when we demonstrated the product would hold during real use. Did the product give the value that we had expected? Was there anything more we needed to do before making the product generally available? This is where we are right now."

The process that he described is a light version of RUP (Rational Unified Process)⁽¹⁾⁽²⁾. It was chosen because the project manager and his team didn't need to invent something new, they believed that it would create working software fast with very little documentation. Being based on RUP, the process itself was designed to be changed and to support changes in the product as normal. Moreover, they wanted the product to grow and the team that developed it to be able to grow in many dimensions: size, geographic distribution, customers, etc. These are some of the properties I require of an agile process.

As the title of this paper suggests, I am a strong believer in an agile process. No one can afford to develop software following a predefined process slavishly. Instead the project should follow a process that helps them to achieve the goal of the team but that project may be described using a predefined process. The key is that you continually adjust the process as you go by reflecting on what you're learning, what works, and what doesn't. In this paper, I will discuss some other properties of a good process that go beyond "traditional" agility.

A process should empower you to focus on creativity

I believe that software development is the most creative process that we humans have ever invented. I also believe that processes or tools can never replace true creativity. However, not everything we do when developing software is creative.

In every project a large portion of the work done by developers is not genuinely creative – it is tedious routine work that we should try to automate. The problem is that the

creative and the routine work are intervened in micro-steps, each such step only lasting from maybe tens of seconds to tens of minutes. Since these two kinds of works are interleaved the developers may still have the feeling of being creative, but fundamentally is not.

Some of you would probably argue that, in fact, you don't do the unnecessary work. You focus on solving the business problem and ignore much of the other work that does not deliver business value. This is of course partly true.

However, even in projects where the focus is on code, people have different approaches to good coding standards. What most people thought was good code some years ago is considered bad code today. Thus people spend time on arguing on these things. With a strong, good leader, these problems may be smaller. However, most teams won't have such a leader.

Aside from process issues we also spend considerable time on many other small, technical issues. For instance, if you want to apply a pattern there are many small steps you have to go through before you have instantiated the pattern. These small steps could with proper tooling be reduced to almost a single step.

I don't think anyone would argue about that some portion of the work that developers is doing is non-creative? I have discussed how large the portion is with colleagues from several camps. Based on my own experience and these discussions I believe it is as large as 80%. That number assumes a light programming environment. There is no significant difference if you develop with or without reuse of existing components. Reuse, may reduce the total work, but it doesn't significantly reduce the non-creative portion of the work. I have only seen one very old scientific study on this subject and would welcome one based on current practices and tools.

To make software development truly agile we need to attack and reduce this 80% of the work effort. We may be able to get from 20/80 (20% creative content and 80% non-creative) to 80/20; I don't think we can eliminate it entirely. The way to attack it is to understand *in depth*, not just on the surface, what is not genuinely creative. Then, having understood it, we can eliminate it by training, mentoring and proper software tools. We will see improved or new tools that also will allow the developers to communicate and collaborate creatively supported by the tools to verify that they are consistent, complete and correct.

If developers can rely on such tools eliminating much of their not genuinely creative work, they will be in a much better position to maximize the time they spend on developing business solutions. This use of tools will also increase the quality of the developer's life. They will be able to spend more time doing the creative work they enjoy; they will spend less time on the routine or repetitive work they dread. This would enable the agility we all are looking for. It would empower people to do the maximum creative work.

A good process allows you to learn as you go – without slowing down the project

Developing software has never been as hard as it is today. You as a developer need to have a knowledge base that is larger than ever before. You need to know about operating systems, database management systems, programming languages and environments, system software, middleware, patterns, object-oriented design, component-based development, distributed systems. You also need to know a software process, a modeling language, and all kinds of tools. And if you succeed in learning something, you can be sure it will soon change. Change is the famous constant in software!

There is simply no way you will be able to learn all this before you start working in a project. You have to learn a little before, but most of it you will need to learn as you go.

The way we learn is very different from individual to individual. Some learn different things better than others. We learn at different speeds, and in different ways. Some people learn better by reading, but others by listening.

Who will teach us? Teachers give classes and mentors participate in projects with the role of being available to help the team while going. As we know, not everyone is a good teacher or a good mentor and even fewer can be both. Good teachers and good mentors are very valuable but not easy to find.

Since we are all different we should ideally be allowed to learn in our own way as we go – from teachers, from mentors or from reading. Traditional education has consisted of attending some basic classes, possibly reading a text-book and then learn by watching or being instructed by a more experienced person.

Just taking some classes on different subjects, for instance in using a programming environment, in a methodology, in testing is a poor start. Software development is much more than that. Going away to different classes which are *not correlated* with a bigger picture – such as a software process -- is not efficient.

Learning from a more experienced person is very effective, particularly if that person is a mentor. However, it is hard to get good mentors – competent individuals with unusual personal skills. There is always the risk that a good mentor may be given other project responsibilities. Time pressure may force the mentor to leave his role of just that – being a mentor. Moreover, someone taking on the role as a mentor may drive into inventing his or her own process, which certainly will slow down the project.

Thus having a mentor on site (maybe only part-time) is a good investment, but we need to reduce the risks I just mentioned. What should we do?

In 1986 I had a dream. I wanted to create a knowledge base, not just a book², for everyone that could play a role in a modern software project: component-based, object-oriented, use case driven ...you name it. Whereas a book may be a nice introduction, it

² I feel that books (like my own books) can give no more than an overview of an approach, but less than the entire story. They don't enable the team to do the entire job. There is much more to it than that.

cannot possible give depth in all areas. I also wanted this knowledge to grow as we learnt more, to change as we better understood how to develop software – throw away what was bad and incorporate new, good experiences. Moreover, it should be able to adapt the knowledge to new technologies as they became available. Since then new, modern technology has exploded – the Internet, J2EE, .NET, new middleware, and much more. Later versions of the knowledge base should accommodate such innovations. In other words what I had in mind was to derive knowledge equivalent to 15-20 books dealing with relevant subjects like requirements, architecture, design and implementation, user experience design, business engineering, testing, software reuse, configuration management, and project management. For ease of use, this information would be written in a common style with consistent terminology. Of course, no single individual would read all of these book equivalents right off the bat. Rather, he would get an overview of the whole approach, she would understand how to go from a business problem to a deployed system. They would have the base on which to grow when they would go into a project that already had started.

To make that dream a reality, the Objectory⁽³⁾ process was created. Objectory grew for more than ten years and evolved into RUP in 1998. What I just described has become a reality in RUP.

With RUP you can go into any depth you wish. It is possible for you to skim the surface in learning about a particular topic or it is also possible to delve deeper to get very detailed information. It all depends on your situation and needs. You have it at your fingertips, just in time. You can often avoid having to ask questions that no one has the time to answer, and can avoid repetitious dumb questions which consume the time of good mentors. You don't need worry about being disciplined by the teacher/mentor (shades of second grade) until you do it right. Is RUP a light process? No, RUP is very rich (on content), but you can use it in a very light way, I will return to this point in the next section. Moreover, since you can learn as you go without slowing down the project, it is an agile process.

However, the authors of the RUP books will never win the Nobel Prize ☺.

A “good” process allows you to be fast – without having to reinvent the wheel!

For years I have claimed--with perhaps a twinkle in my eye -- that the fastest way to create software is not to develop anything yourself but to reuse something that already works. This approach is not just very fast. It is also cheap. It delivers software that works. In practice, in many situations you may still need to develop something new, at the least the glue between the components that you can reuse.

We don't develop our own operating systems, database systems, programming languages, and programming environments any more. We usually start with a software platform and some middleware as a base--not much more. However, much more can be reused:

- You shouldn't have to reinvent a process. Instead, you should use a well-proven process designed to be reused. This is what we call a process framework. A process

framework is both generic and specific. It is generic by being based on a number of sound best practices; these best practices must (if applicable) be integrated and provide a common vocabulary and presentation. The process framework is specific in that it can be successfully specialized or tailored to any application area, to any type of product (pacemakers, claim systems, auction systems, banking systems, command and control systems, etc.), and to any platform being used. It must also reflect the competencies of the developers, the maturity and size of the organization, and whether the work is distributed or not.

Since the process framework is generic it contains more activities and artifacts than any individual development team would apply. For a particular software product you create a process by picking and choosing from the framework.

This is what RUP is about. It is a process framework from which your team with the help of special tools can create your own process.

- You shouldn't have to reinvent the same software over and over again. You should use a process that helps you harvest components and to incorporate existing components – legacy systems or other reusable components – into your design. Of course, with appropriate tools to do so.

What is unique about this process-framework approach is that you can get a product team quickly up to speed. You may still need a mentor who assists the team as it goes. And you still need competent people in the team. The process doesn't replace them, it empowers them.

Before the project starts the team has agreed upon what needs to be done. Once you start, the team can focus on solving the business problem instead of having never-ending discussions on what needs to be done. That just slows down the project. These practices make the framework approach an agile process.

An good process uses tools to do more by doing less

Whatever you do, in order to be efficient you need good tools. Good tools are tools that are developed integral with your process. The process and the tools go together. For instance, in examples taken from another world, if you want your carpenter to drive nails in a wall, he needs a hammer, not a screwdriver. If you want your baby to eat on his own, give him a spoon, not a knife. The same goes for software.

We obviously need tools for programming (coding, debugging, compiling, etc.). Let me call them light tools (even if I can hear many people object to calling these tools light). Light tools must go with a light process to be agile. We have been using light tools for more than 40 years now and got “implicit” requirements, “implicit” architecture, “implicit” design, self-documenting code, we tested like hell. “Implicit” refers to all the diagrams we drew on the white board that were thrown away by the cleaner ☺. Parenthetically, I can't understand given that you have good tools why you would throw

away something that you needed to create the first release of a software product. What you needed for that release you will probably need even more for the next release.

Sure, there were some successes-- thanks to the incredibly good people that made them happen. However, incredibly smart people are a scarce resource.

Moreover, projects that didn't succeed were in the vast majority.

If the software community ever is going to be more efficient – developing good software fast with the desired quality – we need tools to help us do more by doing less. Using only light tools will hold us back where we have been all these years.

I agree with the idea that we want a process that is very light. In fact, I want a much lighter process than I have heard people speak about. It will be light because tools will do the job – you will be doing more by doing less. I want much more in my tools than the light tools that go with a light process. Thus, my tools are not really "light" tools, and to tell the truth my process is in reality rich. However, thanks to my not so light tools – my powerful tools -- that go with the rich process, the process will be *perceived* as very light. And that is all that counts, as long as the process is still agile.

These tools were part of my 1986 dream. The process I envisioned would be that rich process that would go with powerful tools –yet be perceived as very light. The process would be both rich and light. Its tools would help you to create requirements, use cases, architecture, design, tests--everything a developer does. What you would create is not documentation, it is models. When you created a use case for instance, it would be part of a model that actually would become a collaboration among objects and eventually code and test cases. With proper tools we would get that “the models are the code”, or “the code is the models”. Of course, all those work products would be explicit. They would not disappear as the white-board drawings did when the project was over. You would have them for the next project.

Thus, the tools would do the job for you. Once you have created the architecture (in the form of the architectural baseline), you also have executable code. Once you have created the design you have 80% or more of the code that implements it– without having had to think about it. The majority of your test cases follow from your design. Integration tests are created from your use cases. And on we can go. You use blueprints to express your architecture and your design in a way similar to what engineers have been doing for hundreds of years. We have a standard for making these blueprints – we call it a modeling standard (UML). Of course, you use code, too, but just to express what is best expressed by code, namely actions (or operations, methods).

Does this sound like magic? Well, we do some of this today. We get better with every release. It is part of our vision for the future.

We use RUP, UML and tools to support it. We have created light specializations of RUP, which are supported by light tools. There are even lighter versions, but we will not suggest that you water down the process. Rather, we will convince you to keep our best practices: develop iteratively, manage your configurations, find the right architecture first

– refactor when you didn’t succeed, etc. Within these invariants your team can create the process it wants. It will be a rich and light process, a process that is perceived to be light even though below the surface it is very rich. On top of all this, you can scale up your process to a larger team, to a team of teams distributed around the world, and to a product with many different features allowing a lot of customization. Still your light version will not and shall not be burdened by this ability to scale up. We call this “right-sizing” the process.

An agile process relies on tools for whatever you do from “womb to tomb”. An agile process is rich and light -- perceived to be light, even if it is very rich. It is thanks to the rigorous tools that go with the rich process that the process becomes light. You will be able to do more by doing less. And there is more to come.

Look out for the next big thing

A process framework with tools significantly empowers you as a developer. The knowledge base about good software that comes with RUP continues to grow, the process framework becomes easier to understand and to use, more and better tools evolve. You will be making this effort.

However, many others will not make these investments. They will jump for light processes with light tools and assume that good people can work wonders. They will rely on light development practices that have been around under other names for decades. The problem is that software development will still be hard. In fact, it will become even harder. We will see many software companies fail. For those others that won’t make these investments, there is to my mind only one way out that dramatically – not just marginally -- changes this picture. It is the next big thing.

In 1980 I wrote a letter to the president of Ericsson, the essence of which was that the component-based approach we were then using would evolve into a world standard. We should now go forward by 1) developing our modeling language and 2) our process into a world standard supported by tools, and on top of process, language and tools, 3) developing expert system support.

I realized that the expert systems could not be powerful enough to do the job, 1) before we got a standard modeling language like UML, 2) before we had a knowledge base like the one in RUP, and 3) before we had supporting tools. Twenty years later we are there; we can take the next big step. It is “intelligent” agents.

We can put a layer of agents on top of RUP and its tools. These agents are programmed to trigger for different events. They recognize patterns and anti-patterns, suggest resolutions. They assist the developer dynamically in using the knowledge base. They suggest the next micro-activities to be performed, many of which follow from the context of the work being done and do not require anything creative by the developer. The developer is empowered to “think” and not be bothered by all the small pieces of work that today constitute what we usually call “the hard work”.

The process will be very agile. It will be perceived as a very light process, but it is based on very rigorous tools. Obviously, you learn as you go. It can easily adapt to different roles that people play and it can scale up as needed.

Does this "next big thing" sound like science fiction? Maybe, but I don't think it is far ahead⁽⁴⁾. Am I saying this because I am personally involved in it? I can understand that I risk my credibility, but I have to take this risk. I wouldn't support this without believing in it and that we can get there soon. And we can only get there thanks to the existence of UML, RUP and supporting tools.

Let me conclude by saying that although I have in this paper primarily discussed processes from an agile perspective, we expect much more of a process than just being agile. We want it to empower you to solve your business problems and to satisfy your users. We want your systems not only to work 7x24, but to grow gracefully as the world changes. To get there you need a process framework, a modeling language, and supporting tools, not the least of which will be the "next big thing," intelligent agents.

⁽¹⁾ Kruchten P., Rational Unified Process – an introduction, 2nd ed., Addison-Wesley Longman, 2000.
see also <http://www.rational.com/products/rup/index.jsp>

⁽²⁾ Jacobson I., Booch G. and Rumbaugh J., The Unified Software Development Process, Addison-Wesley, 1999.

⁽³⁾ Jacobson I, Object-oriented Development in an Industrial Environment, Proceedings of OOPSLA'87, Special Issue of SIGPLAN Notices, Vol. 22, No.12, December 1987, pp. 183-191.

⁽⁴⁾ See www.jaczone.com