Model-Based Translation: The Ultimate Agile Process

H. S. Lahman

Agile software development is an approach to constructing software that is designed to be responsive to change in the business environment. In particular, agile development allows rapid reaction to changing requirements. Associated with agile development is the idea of getting working software into the customer's hands quickly. This allows rapid feedback on acceptability criteria and, under some circumstances, earlier time to market.

While agile development can be associated with any sort of software, it is most commonly associated with object-oriented development. More specifically, the most common and well-publicized applications of agile techniques have been focused on software development using object-oriented programming languages (OOPLs) or OOP-based development. The purpose of this paper is to examine an alternative to the OOP-based processes that is based upon model-based development; specifically one object-oriented analysis (OOA) models. The primary goal is to demonstrate that model-based development can actually be more agile with respect to the primary goals of agile software development than the OOP-based processes.

A secondary goal is to respond to some unfortunate hyperbole associated with the OOP-based agile evangelism that casts any development involving software modeling as BDUF – Big Design Up Front. This erroneously tars all model building as moribund bureaucratic activities associated with massive, monolithic waterfall development cycles. It will be demonstrated that this is not true at all and model-based development can actually provide far more agility than the OOP-based development processes.

Background

Before going into how model-based development can be agile, though, it is necessary to provide some context for the discussion, both historical and technical. Since software construction is a dominantly intellectual activity, it is useful to distinguish between *methodology* and *process*. In the dictionary sense these words are close to synonymous. However, in a software development context they can be used to characterize quite different aspects of software development. So in this paper the following definitions will be used:

> ***Methodology***. This is a suite of guidelines, practices, conventions, standards, policies, and whatnot that guide the creation of the intellectual content of software construction. Thus a methodology guides the traditional design activities normally associated with OOA, OOD, and OOP.
>
> ***Process***. This is the more traditional notion of an ordered set of steps that are required to design, construct, and deliver a software product. Though the activities associated with some steps are inherently methodological, most process activities tend to be rather mechanical and rigidly defined. Thus the process is a framework that defines the sequence of development and some of the more mundane activities.

It is important to understand that agility is primarily a process issue. From a methodological viewpoint agile and non-agile processes typically employ the same intellectual design activities because both are driven by the OO paradigm. The thing that most clearly distinguishes agile from non-agile development is the context within which the intellectual activities are organized. That is, the way one performs OOA, OOD, and OOP is largely unaffected by whether the surrounding process is agile or not. (There are some methodological differences between the OOP-based and model-based approaches, such as the nature of refactoring, but they tend to be comparatively minor and they are mostly tied to the nature of the work products.)

The traditional process for software development, dating from the ‘40s, has been elaboration. The basic idea behind elaboration is that one manages complexity by incrementally refining the software design through a set of fixed stages, such as OOA, OOD, and OOP. Thus the OOA describes the solution from a purely customer view and addresses only functional requirements. Then OOD elaborates on the OOA solution by resolving nonfunctional requirements at a strategic level. Finally, OOP elaborates on the OOD solution by providing tactical solutions for the particular computing environment. At each stage design methodologies provided intellectual content that addressed different issues. That separation of concerns allowed better focus on the issues at hand, such as functional requirements in OOA.

Much of the computing space is highly deterministic because it is necessarily based on solid mathematical models. Such determinism invites automation. Automation was initially reflected in the evolution of things like Assembly Language, linkers and loaders, and 3GLs. However, in the early ‘80s automation began to advance very rapidly. Sophisticated operating systems, “canned” infrastructures, GUI builders, high end DBMSes, web site builders, and RAD IDEs all flooded the computing scene. At the same time an even grander scope for automation was launched when translation techniques became an alternative to elaboration.

In translation the basic idea is take advantage of determinism and automate as much as possible of the computing space. The OO paradigm was becoming popular at that time and it was ideally suited to translation because to the separation of concerns of OOA, OOD, and OOP and because OOA solutions were already independent of the computing environment. In effect translation techniques sought to automate everything in OOD and OOP. Essentially a “translation engine” could automate and optimize OOD and OOP to generate 3GL or object code directly from OOA models. This achieved a form of design reuse in that the intellectual content of OOD and OOP could be applied to providing a translation engine that was reusable for any OOA solution. (One would need different translation engines to optimize for different computing platforms but on any given platform one would need only one translation engine for all applications.)

Alas, the computing space may be highly deterministic, but it is also very complex. So it was not until the late ‘90s that translation engine technology could produce optimized 3GL code that would be competitive with manually developed code in terms of cost and performance. Through the ‘90s, though, another force was becoming dominant on the computing scene. The demand for interoperability, plug & play, and common look & feel in the tools that evolved in the ‘80s grew exponentially. This led to application frameworks and standardization.

One important standard was the advent in the late ‘90s of OMG’s MDA (Model Driven Architecture) initiative. The MDA provided a conceptual framework for the tools that supported both elaboration and translation. Thus the “translation engine” of the ‘80s became the MDA transformation engine. The “translation rules” used to parametrically control optimization became MDA Transformation Models and Marking Models. And the OOA model used as input by translation became the Platform-Independent Model (PIM). Among other things, the MDA facilitated the evolution of plug & play code generators that could work with any UML drawing tool. The MDA also provided a conceptual basis for large application frameworks and IDEs, such as Eclipse. All these things played together to launch translation techniques into the mainstream of software development.

However, translation involves more than just buying or building a transformation engine. It fundamentally changes the way that software is developed. One effectively moves the focus from 3GLs like Java and C++ when doing elaboration to 4GLs like UML where the application developer only deals with high level models. So the application developer provides intellectual content to only one stage, OOA. That change affects almost all of the processes associated with software development, from configuration management to testing. Interestingly, though, the methodologies for constructing software are largely the same between translation and elaboration. The “programming language” changes from a 3GL to a 4GL, but the design activities one applies are pretty much the same once one gets past the notation.

It is essentially the process of translation and supporting tools that enables agile development in a model-based development environment. For example, the design reuse of the transformation engine takes advantage of economies of scale by removing most of the computing space optimization concerns from the application developer. That makes the application developer more productive by delegating optimization to specialists. There are also intrinsic advantages for responding to change when employing a solution

notation that is very compact. In addition, one can apply agile process practices to 4GL development the same way they are applied to 3GL development; its just another programming language. (All these will be discussed in detail below.)

Figure 1 represents a simplistic view of how these various factors play together to provide an agile development environment. In the diagram MBSE represents Model-Based Software Engineering, which refers to both the methodology and processes around translation-based construction. Agility is imposed upon the basic translation approach through MBSE. While elaboration can be entirely manual, translation necessarily requires tool support. The tools are complex and require a conceptual framework that MDA provides. The tools and MBSE combine to provide a bundle of customer benefits.

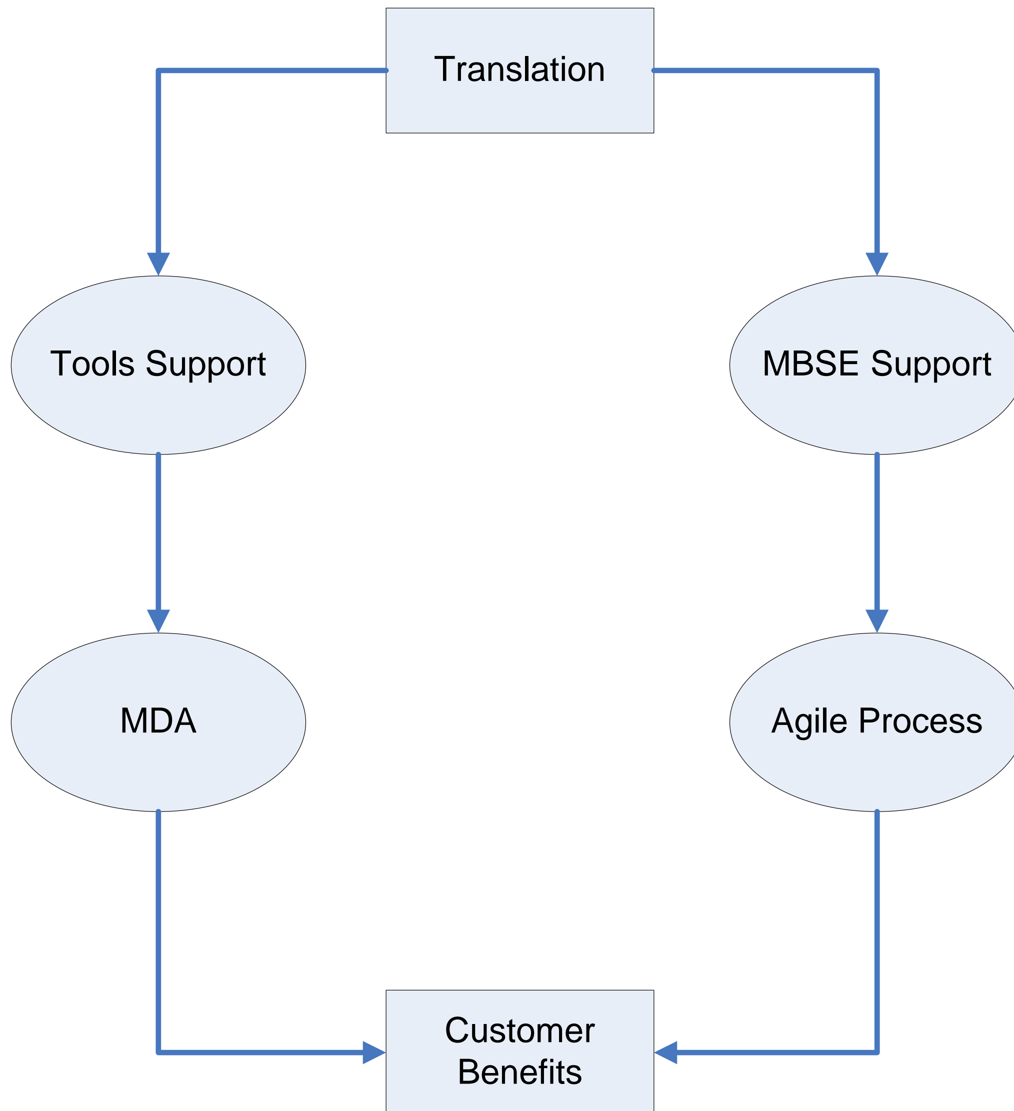

Fig 1. The context of agile model-based development

Though the tool limb and the process limb are conceptually independent, there is some overlap with respect to agility through things like design reuse that are inherent in automation. The rest of this paper goes through the traditional agile practices and shows how they are applied to and enhanced by translation.

Incremental/Iterative Development (IID)

IID is a core element of all agile processes. The IID model dates from the '60s but has been modified somewhat by agile processes. In the IID approach the software is subdivided into smaller, more manageable chunks that are developed sequentially in a quasi-independent manner. (The notion of iteration enters the picture by modifying previously developed chunks to accommodate the new chunks.) Originally IID was applied within traditional waterfall stages[1]. For example, all design work would be subdivided into chunks and developed incrementally, then all designed chunks would be implemented incrementally, and so on.

Agile processes modify this concept by insisting that each increment's chunk of software be fully implemented so that the software could be executed and the customer would have tangible results. This has several advantages. Perhaps most important, it eliminates the "90% Done Syndrome" where large projects proceed until they are 90% done according to PERT analysis and then languish in that state for extended periods of time, perhaps until the project is cancelled. By providing working software for each increment the agile processes ensure that tangible customer benefits are achieved throughout the project development. That gives the customer a much better handle on progress. Other advantages are: facilitated project management; better and more rapid feedback from the customer; early detection of requirements problems; and early detection of fundamental problems in implementing what the customer really wants.

In applying MBSE, one can do exactly the same sort of IID as the OOP-based agile processes. Essentially the only difference is that features are being fully implemented in a 4GL rather than a 3GL. That's because the software features are exactly the same for a model-based process as they are for an OOP-based process. However, translation provides some interesting advantages over OOP-based agility. Since there is design reuse through automatic code generation that resolves nonfunctional requirements, there is less to "design" in a translation process for each feature. That is, the developer is not distracted by pure computing space issues like performance, size, concurrency, and a host of other issues related to nonfunctional requirements. And since 3GL code is automatically generated, the time to actual implement the design is greatly shortened.

A secondary advantage is that in translation systems the models are executable. That allows one to validate the resolution of functional requirements without actually generating production code. (In translation processes one usually runs the same test cases for functional requirements against the models as one runs against the final executable; only the test harness changes.) This allows an alternative view of completion that can be useful in certain situations. For example, the hardware in a hybrid system may not be ready yet for execution but one can demonstrate that the software design will do as the customer expects without the hardware.

A tertiary advantage is that translation techniques in MBSE provide out-of-the-box partitioning of large applications in a disciplined fashion. That results in isolation of functionality in subsystems. In turn, that allows better management of feature sets because features are usually based on narrowly defined functionality. That is, features are likely to be implemented primarily in a single subsystem. Translation tools are designed to process subsystems individually, which makes it easier to isolate the feature for implementation and testing. That also allows parallel development of features by multiple teams without concern for the teams stepping on each other's toes.

Before leaving this topic it is worth taking a moment to dispel a popular myth about model-based development. The OOP-based agile process advocates sometimes disparage model-based development as BDUF that is associated with the monster waterfall development models from the 1970s. Because such process models usually involved a lengthy design stage where design models were produced, the implication is that large, monolithic waterfall development is synonymous with design modeling. *That implication is absolutely false*.

All software development involves some sort of waterfall development model, including the OOP-based agile processes. That's because developing software involves multiple activities that necessarily must be done in sequence. Thus every increment in an IID process is a mini-waterfall. For example, in eXtreme

---

[1] Probably the first formal exposition of IID dates to Barry Boehm's Spiral Model from the '60s.

Programming, tests are defined, CRCs are developed, code is written, tests are run, refactoring is done, and tests are run again – all in a rigidly defined waterfall sequence. The only thing that changes is the *scale* of the waterfall and the mechanisms for iteration, such as refactoring. Thus a development team may choose a very large scale for single monolithic development, in which case one has classic BDUF. But the team can just as easily choose to do IID where the waterfall increments are just a few weeks, days, or even hours. That is quite feasible for model-based development and feature-based IID is already the dominant development paradigm in modeling shops, primarily because it makes project management easier. As described above, it is quite easy to do full development for each increment with a transformation engine providing automation. That enables agility with a model-based process. So there is absolutely no justification for equating modeling with BDUF and monolithic waterfall development.

Customer control of the feature set

In agile processes the customer controls the feature set. In agile processes the customer selects which features will be implemented in the current IID increment. This allows the customer to "steer" the project by changing priorities and substituting features as the development proceeds. The customer also decides when to cut off the IID sequence and release the production version of the software to the marketplace.

So long as feature-based IID is employed and features are fully implemented, this sort of "steering" is enabled. All of these enabling conditions can be satisfied in a model-based environment. In fact, from the customer's viewpoint there is absolutely no difference between an OOP-based agile process and a model-based agile process.

Requirements change management

Another hallmark aspect of agile processes is that such processes are more responsive to changing requirements. One aspect of this is enabled by IID. If the customer "steers" the project by determining what features to include in the current increment, the customer has control over what features end up in the final product. So if there is a change in plans or some uncertainty early in the development, the customer has some control of what feature-level requirements will actually be met by omitting or substituting features in later increments. As a practical matter that has limited value with modern product development techniques because feature sets are usually fixed prior to committing to development. However, any process that employs IID can provide this sort of agility.

A second mechanism for agile change management is the rapid feedback from the customer on the way requirements were implemented. One of the most common problems in delivered systems is that the system is not quite what the customer really wanted or needed. There are many possible reasons for this ranging from faulty requirements specification to faulty implementation. If a problem is related to requirements specification – which it often is – then new requirements need to be generated and implemented. The rapid feedback from fully functioning code in agile processes allows early detection of requirements problems when they can be most easily corrected. It also avoids nasty surprises when the product is finally delivered. As discussed above, model-based processes are quite capable of executing feature-based IID with full implementation to an executable. Thus exactly the same sort of rapid feedback response is possible for model-based agile processes as for OOP-based agile processes.

A third mechanism for agile change management lies in the mechanics of IID and institutionalized refactoring. Any change to existing requirements for software already implemented in previous increments can be added to the current increment as if it were a new requirement. This allows implementation of the change in a systematic way that supports effective tracking and management because of the short duration and limited scope of an IID increment. This is possible whenever IID is employed as the base development paradigm.

Refactoring, though, is an approach to change management that is unique to agile OOP-based processes. Responding to a change in requirements necessarily means that the application needs to be modified. At the 3GL level there are other issues, such as maintainability of the code, that also affect how it is constructed and modified. The agile OOP-based processes have incorporated code refactoring as a fundamental tool. This is primarily driven by the need to maintain the code manually. Refactoring ensures that the code will always be as maintainable as 3GL code can be. That translates into less time needed to

make changes to the code. Thus refactoring the original code to make it maintainable anticipates the need to modify it later.

The problem here is that maintainability is a developer problem rather than a customer problem. In addition, maintainability is hindered by the OOPLs because they are all 3GLs and they all suffer from physical coupling due to comprises necessary to map into the hardware computational models (e.g., procedural message passing, procedural block structuring, stack-based scope, type systems, etc.). Therefore an entire discipline has evolved to deal with physical coupling in OOPLs called Dependency Management. Entire books have been written about how to do dependency management and a substantial fraction of the developer's time in OOP-based agile development is devoted to dependency management. While refactoring addresses change management through efficient maintenance, the price is rather high.

In a translation-based development the developer never needs to manually maintain the 3GL code. If changes need to be made, they are made to the models and the code is regenerated. (In fact, at least one transformation tool vendor deliberately makes the generated code difficult to read to prevent the developer from mucking with it rather than the models.) Therefore all the problems with physical coupling in the OOPLs go away and there is no need for that level of dependency management. This alone results in a substantial effort reduction for translation-based development compared to OOP-based agile processes and that reduced effort translates into higher overall productivity and greater responsiveness to customer needs.

But don't models have to be changed when requirements change? Yes, they do. But there are three factors that make change much easier in a model. The first is that in an OOA model there is no inherent problem corresponding to physical coupling in the OOPLs. OOA is the purest form of the OO paradigm. The OO paradigm is designed to produce maintainable applications. Since there are no inherent flaws due to compromises with computational models, good maintainability comes pretty much "for free" with good OOA modeling practices. Thus the design methodology associated with MBSE directly ensures good maintainability of the models.

The second factor is separation of concerns and design reuse. In the OOA model only functional requirements are resolved and the nonfunctional requirements are handled by the transformation engine. Thus the application developer can be much more focused on the change and its implications in the OOA model because there are no distractions over nonfunctional requirements. (Typically 40-80% of the 3GL code in an application is related to computing space issues rather than functional requirements.)

The third factor is the compactness of the model notation and support of multiple views of the solution. The equivalent 3GL code will be at least an order of magnitude more verbose than all of the UML diagrams and abstract action language code that comprise an OOA model combined. That alone makes it much easier to isolate change and evaluate its impact. Almost as important is that an OOA model offers several quite different views (e.g., static structure vs. dynamics) of the solution at different levels of abstraction. That greatly facilitates determining what needs to be changed, determining how it needs to be changed, and evaluating side effects.

Until one actually does maintenance via OOA models, one cannot comprehend how much faster and more reliable it is than maintaining OOPL code. Similarly, once one has used a good model-level debugger, one never wants to go back to a 3GL debugger. One regards the VC++ debugger the same way a VC++ developer regards an Assembly debugger; life is just to short for that.

Integration of testing

An important characteristic of agile processes is the integration of testing within the process itself. This is a major break with traditional development where testing was something that was done after all the development work was completed. While this doesn't directly contribute to agility, it is worth examining with respect to model-based processes simply because it is so fundamental to OOP-based agile processes.

The OOP-based agile processes employ essentially a two-tier approach to testing: unit testing and acceptance testing. In theory acceptance testing is the customer's responsibility but it is typically delegated to an in-house SQA group. In the OOP-based agile processes tests are the only formal statement of requirements because these processes typically employ purely verbal requirement elicitation from the customer. So the ultimate arbiter of whether the software correctly implements the requirements is the customer's acceptance tests.

In addition, a form of unit testing it incorporated in the developer's daily work. This is mainly verification that the developers have built what they thought they should build. By integrating such testing into daily work – down to the code fragment level – the developers are forced to do something approaching reasonable testing. [A common problem for traditional development is burn out at the end of a project. At that point developers want to be done with the project so <unconsciously> they don't want tests to fail because then they will have to fix the problem. This is a subliminal conflict of interest that tends to lead to poor testing and, consequently, lots of field escapes. By integrating testing into daily work the burn-out problem is avoided.]

The unit tests done in OOP-based agile projects are not traditional unit tests. They do not completely isolate the unit under test (UUT). Instead they test the same very limited functionality in situ. (This is only possible if each increment produces working code.) The advantage is that the unit tests are "re-testing" other functionality already implemented so that one also gets a sort of functional system test as well. Another advantage is that no special state initialization or stubbing is required so tests tend to run faster and require less work to create. The disadvantage it that this isn't a true functional test because there is no attempt at systematic path coverage, other than in the UUT. Thus the only true system functional tests are the customer acceptance tests[2].

Another, relatively new, aspect of testing in the OOP-based agile processes is Test-Driven Design (TDD). The basic idea here is that one writes the tests first and then designs the software to pass the tests. TDD provides a methodology for such test creation that helps to ensure the software will be constructed properly[3].

As it happens, the translation-based approaches routinely provide three levels of testing: unit, subsystem, and system. Most translation tools are designed to process subsystems independently. That includes built-in infrastructure to support testing at the subsystem level. The MBSE modeling paradigm also requires "firewall" interfaces between subsystems (i.e., pure event-base interfaces), which decouples subsystems implementations. This makes it possible to do full functional testing of subsystems on a standalone basis or even in groups. This is a very powerful capability for large systems.

In addition, the use of state machines to describe object behavior makes it trivial to conduct traditional unit tests at the state action level. One can completely specify a test in terms of: (A) an input event; (B) the state variable (object attribute) values before processing the event; (C) the state variable values after processing the event; and (D) any events generated during the processing of the stimulus event. It is trivial to modify the event queue manager so that it simply logs events that the UUT produces once it pops the initial stimulus event. Thus one can exhaustively test any method in a subsystem when the only behavior implemented is that method. (Instances with relevant attribute values do have to be instantiated.) This is possible because the event queue manager, combined with the asynchronous communication model of state machines, decouples the method implementation from other methods.

As indicated above, the translation tools typically provide infrastructures to support automated testing. This saves significant effort for manually providing such infrastructure in the OOP-based agile processes. Since test specifications can be produced any time, there is nothing to prevent a model-based process from

---

[2] This is potentially a major problem because customers usually have expertise in how to write good tests. If acceptance tests are delegated to an expert SQA group there is a second problem. The requirements need to be communicated verbally by the customer to both the developers and the SQA people. That leaves room for misinterpretation, inconsistency, and other problems that won't be detected until the acceptance tests are executed. (Any lag will be mitigated if acceptance tests are run for each increment, but it may still be awhile before the customer recognizes deficiencies.)

[3] Unfortunately this view reflects a view of product quality that has been discredited since the '80s in manufacturing. The basic assumption is that product quality can be determined through testing by eliminating defective units before they are released to the field. The OOP-based agile processes actually institutionalize this view by mandating that the only formal requirements are specified in tests. In fact, the best that testing can typically hope to provide is something on the order of 5-Sigma reliability. To achieve 6-Sigma and beyond one needs militant process improvement and defect prevention techniques. However, it is beyond the scope of this paper to go into this issue more deeply.

creating tests up front, executing them with each increment, or even employing TDD. That is really just a matter of development policy for the shop.

The fact that translation OOA models are executable provides another unique feature for model-based testing. One can execute exactly the same test suite for functional requirements against the models and production executable. As mentioned previously, this provides a more versatile way to approach validation that can be very useful for hybrid systems where the hardware is being developed in parallel to the software. An added benefit for R-T/E shops is that when hardware integration testing finally does take place, one is never in doubt about whether a problem is due to hardware or software. That's because the software has already been validated for the test suite.

Customer interaction

The agile processes provide responsiveness to the customer in a variety of ways. Some of these, like feature set "steering" and rapid feedback for working software, have already been discussed. In this subsection the aspect in focus is requirements elicitation. In the OOP-based agile processes this is primarily a verbal process and the customer (or customer agent) is expected to be available on a day-to-day basis for the developers. This works because of short increments that produce working software. And miscommunications are readily identified by the customer when the increment's executable is delivered and that allows rapid corrective feedback.

There is nothing to prevent providing requirements this way in a model-based environment using agile IID but it tends to be uncommon. The reason is that model-based development's advantages are best realized on large projects with several subsystems and large teams. Large teams are typically broken down into smaller teams to work in parallel on different subsystems more or less independently. However, to do that the application needs to be partitioned properly and the requirements allocated to the subsystems. That involves more communications for things like negotiating subsystem interfaces. In that sort of situation one usually needs to provide more formal, written documentation of requirements. In other words, the OOP-based agile processes are designed for small projects that can be entirely done by a small (<10) team[4].

Pair development

This is another unique characteristic of OOP-based agile processes that does not directly affect customer response but is distinctive enough to warrant consideration. Pair programming provides an integrated review process for code because four eyes are better than two for recognizing problems. When pair participants are rotated, as they usually are, it also provides communication of best practices that should be shared by the entire team.

While using pairs to develop models is certainly feasible, it is very rarely done. That's because models are usually developed on a consensus basis as a full team effort. Some models, like State Charts for object state machines, are done individually and reviewed by the full team. Such models are small so they are easily constructed and reviewed. The overall behavior and event sets, though, will usually be defined as a team effort at the Class Model or Interaction Diagram level.

There are several reasons why full team activities are favored in a modeling environment. One is the compactness and level of abstraction of the models. That enables focused communication. Similarly, the different views of the solution provided by the various UML diagrams and action language foster better focus and communication. Finally, the narrowing of scope to functional requirements in the OOA model also increases focus. That all makes reaching consensus with a larger number of people easier so the efficiency of larger groups is improved. In turn, that makes the obvious advantages of wider review attainable with minimal cost in effort.

---

[4] The OOP-based agile processes can be used for larger projects but a Systems Engineering function needs to be added for application partitioning and coordination. That is provided out-of-the-box with MBSE but not with the OOP-based agile processes.