Chapter 3
Types of Processes

"There is more than one way to skin a cat, and the approach you take depends on how much of the cat you want left. It also depends on whether you're after the cat or the skin."—Anonymous

Just as the earlier editions of this book were targeted at a specific group of developers, they were also targeted at a specific type of process—the so-called waterfall method. But not all development fits into this one method, any more than all developers are the same or all development is the same. In this chapter, I'll discuss what your options are for developing a custom software application.

The processes for software development make up a spectrum not unlike the color spectrum. In the color spectrum, colors evolve from red to orange to yellow to green to blue. However, there is no specific point that you can define as "blue" or "yellow"—there are many shades of yellow, some that are full of orange, and others with a markedly greenish tint. Nonetheless, if I tell you that I'll pick you up in a yellow car, you know roughly what to look for. My car might be a bright yellow taxicab, or a washed-out-yellow Civic, or a mustard-yellow Corvette—but you know I won't be showing up in a black Grand Prix or a white mini-van. The term "yellow" defines a range of possibilities.

Similarly, the software development processes range from highly structured methodologies at one end to free-form, every-man-for-himself cowboys hacking at the other. In between the two end-points are types of processes—Rapid Application Development and Extreme Programming, to name two—that, like colors, don't identify an exact point in the spectrum, but rather serve as markers for areas along the spectrum.

Different processes have different requirements, both from the customer's point of view as well as with respect to the developer. As the software developer, you'll be charged with determining which type of process is most appropriate for a given situation. In order to do so, you need to be familiar with what the possible processes are in the first place.

Structured development (the waterfall method)

Structured development, or the waterfall method, gets its name from a visual interpretation of the process. A waterfall starts in one place, and then descends to a second level, and then a third, and then a fourth, but never goes back to a previous level.

The waterfall method of software development follows the same path. It starts out with an analysis of the problem, determining the requirements of the solution, and then moves on to a design of the solution, then the development (programming) of the design, then the testing, repair, and modification based on the testing, and, finally, the implementation.

In many cases, each of these functions is performed by a different person; indeed, many times by separate groups or departments. Each step must end before the next one begins, and the customer must sign off on successful completion of the stage via a formal acceptance process before the next step begins.

This process came about early on in the computer age because it was the only possible way to develop software back then, and because the people doing it were trained in other disciplines and had backgrounds in other industries where this process was the norm.

The first programmers were actually hardware engineers who needed to develop software so they could test their hardware. They were used to the analysis/design/build/test/implement cycle because that's how they built the hardware.

In addition, the CPU cycle was precious, and the tools for iterative development didn't exist. You simply didn't have the luxury to slap together a program and run it, counting on the compiler to catch your bugs and errors in design—you needed to take care of those things before ever submitting your program to be run.

The advantage of the waterfall method is that it can produce solid, reliable applications that do what they are designed to do, particularly for applications that are relatively static. Furthermore, given an experienced team with a history of development, the time and dollars required to build an application can be determined with reasonable accuracy once the specification has been completed.

There are four primary downsides to the waterfall method. The first is that it takes a long time, and for much of that time, no apparent progress is made. The analysis and design meetings can seem to take forever, and then, once programming has started, there aren't many deliverables for a while.

The second is that the waterfall method doesn't provide much flexibility for change. Once the design has been completed and programming has started, modifications as a result of new requirements being discovered can't simply be added on the fly. Ordinarily, modifications are handled through a formal "Change Order" process that in and of itself can be time-consuming.

However, over time more and more tools have become available to help complete various portions of the process, and it seems that the more automation that's been introduced into the waterfall method, the more willing our users are to go back and change something that was done in an earlier phase. So, in a way, these automated tools have enabled a type of iteration that wasn't practical years ago. For example, it is easier to go back and change a database structure because of the use of some CASE tool, so 1) the team is more willing to make such a change, and 2) the customer is willing to do it because it no longer has such a negative impact to cost and schedule.

The third downside is that the waterfall method requires a level of discipline and sophistication that is hard to imbue in the PC culture. The personal computer enabled rapid iteration of projects—but since they were, in the beginning, "small" projects, they escaped notice of the formal disciplinarians in software development. Then PCs became powerful enough to do "real" applications, and the industry responded by trying to put processes that were good for mainframe projects onto small "PC" projects. When we realized we needed something, we slapped on that proven old waterfall method—now we are trying to find a happy medium where relatively small projects follow a process, and that process ideally can enable the team to meet the needs of the e-economy pressures and needs but not burden it with mountains of administration.

The fourth downfall is that the structured development process can tend to shut out users from partaking in the development process, and thus a sense of ownership isn't fostered as well as with other methodologies.

Note that many people refer to the waterfall method as "structured development," and that's how I'll refer to it in the rest of this book.

Rapid Application Development (RAD)

In a formal sense, "RAD" was coined by James Martin in the 1970s as a result of his research on software engineering. It refers to a specific approach regarding application development. In the three decades since, though, the term has been mutated in every conceivable way to the point that it's unrecognizable and virtually useless, except as a generic moniker for "cool and fast" by marketing folks.

In a rough sense, though, we can use RAD as a catch-all term for the next step in the evolution away from the waterfall method of application development. In the traditional waterfall approach, users are consulted only during the initial analysis phase. Design, developing, testing, and deployment are executed in series, and the users don't see the developer's work until the application is deployed. This, of course, makes it difficult, and very expensive, to implement changes as well as compensate for changing requirements, miscommunication, and other mistakes.

Instead of waiting to finish one phase of the development process before proceeding on to the next, what if you sketched out a rough design, and then built a prototype that acted as a model for the design? And you incorporated user input and feedback during that process? Then, with that prototype, you'd gather more formal feedback, and go back to the analysis and design phases, and improve them. You'd retool the prototype, incorporating the feedback and new features in your design.

This process, iterating between analysis, design, and prototyping, would continue until you reached one of several milestones—the improvements and changes being suggested in each pass would gradually become smaller and smaller until they could be ignored or postponed to another release, or until you ran out of time or money and had to move to programming and implementation.

The prototype application becomes an aid to analysis and design by giving the users a concrete basis on which to evaluate the performance of the application, rather than abstract diagrams and descriptions.

Depending on the type of application being built, a significant portion of the programming may have already been done during the prototyping as well.

Benefits of RAD

The prime advantage of RAD is that you get a product that is much more likely to match the users' requirements because they get to "see and touch" their application—actually working with a prototype is much different from looking at screen shots or text-based descriptions. It is also possible to get an application up and running more quickly than with the traditional waterfall method because the analysis and design can be done faster with the users' help during the prototyping, and because some of the development can be done in parallel.

But there's much more to it than that. Let's investigate the many benefits of RAD.

Customer ownership

When your users are working alongside you during the development and prototyping of the system, instead of simply listing requirements, watching you ride off, and wondering what you're going to come back with, and when, they're much more inclined to take ownership of the system.

A number of years ago, I built a system that was going to replace a DOS-based application that the company's sales reps had all been using for eight or nine years. The old system had been developed in house by a developer whose office was right down the hall from the IS director. The system was widely liked in part because it was "their" system—not something bought off the shelf or otherwise acquired.

When the company decided to contract out for the replacement, there was some nervousness as to whether the system would be as well accepted by the reps. As a result, the IS manager and the VP of sales both participated in RAD meetings every Friday throughout the winter and spring as the system was being prototyped.

When questions later came up about "the system," the two were quick to mention that it wasn't just "the system," but "our new system." The two, in fact, took some delight at customizing and enhancing pieces of the system, schmaltzing up demos for grand announcement, and acquiring doo-dads that went along with the system, such as mouse pads with the system's logo and golf shirts designed for each sales rep with the name of the system above the chest pocket.

It was very clearly "their" system even though an outside shop had developed it.

You will likely find that it is easier to engage their participation—you may even find that you can get them to do legwork more easily, because it's *their* system.

Knowledgeable users

In the same vein, when users are involved in the development process, they more quickly and more fully internalize the functionality because, after all, they were the ones who determined what that functionality would be, and had a direct hand in determining how that functionality was going to be implemented.

In the system mentioned in the previous section, the IS manager and VP of sales both led internal training classes for the sales reps and sales support staff. Since they were intimately familiar with the system, they were fully capable of showing others how to use it. And they're leading the class.

Depending on your user community, you may even be able to find people who will actually *read* your specifications and documentation (gasp!), and write the final user documentation as well.

Better testers

The third advantage to using the RAD process for this particular system was that these two individuals were also the initial beta testers, and they were able to catch bugs that involved major functionality immediately. For example, one of the modules had a price calculation screen that involved a number of very complex calculations. Since they both knew the business that the system was modeling and the system, they were able to match up expected results with actual results for a wide variety of scenarios—something that a casual ("Oh, we gotta do some more stupid testing today") tester would likely not have been able to do as thoroughly.

You'll likely find with your users that since they have taken ownership of the system from the beginning, it's in their own interest to make sure it works properly. Think about how you treat the acquisition of a brand-new car of your own vs. when you get in a rental car. You're much more likely to examine every knob and button before you drive off in your own new car. With a rental, you'll adjust the seat, make sure you know where the essential controls are, and

then take off. Similarly, you're much more likely to park in an out-of-the-way spot with your brand-new car than a rental that's a couple of years old.

Same thing here—since the system isn't simply being "dumped" on them, they're going to take care of it—starting with making sure it works properly from the get-go. They'll be involved in the user acceptance testing from the start—including helping write the tests to make sure they fairly stress the system.

Multiple champions

When your users are involved from the beginning in the design and development, you'll likely not have to work as hard to "sell" the system to the user community because having people involved along the way will generate excitement. Part of that excitement will translate into your acquiring additional champions.

As I said in the beginning, you should always have a champion for your system in the user community or at the customer's site. If you're the only proponent for the system, it's near a sure bet that the first instance of complication will cause enough trouble for the system to be waylaid or abandoned altogether.

In the opposite fashion, the more champions you acquire along the way during development, the more likely that problems you encounter will not cause a derailment of your system.

Reduced fears

As the number of users who buy into the system increases, the number of fears in the user community in total goes down. Not only will the new converts not have the fears that typically accompany a new system, but the remaining "unconverted" will have their fears lessened as well, since they'll see a larger and larger number of people who are not worried about the impact of the new system.

Ease of data migration

If your system requires any sort of data migration, that process is more likely to be successful if you have users who can bridge knowledge of the old and new systems.

I have never run into a customer who actually understands what their current data looks like, or truly knows what's in it. There's always garbage in some fields, legacy fields that have never been removed, and even whole sets of garbage records that end up hanging around. As a result, if they don't truly understand their data, how are *you* going to write a conversion routine that actually works? You're going to need their help in making those decisions about how to deal with fields that contain unexpected or invalid data as well as records that shouldn't be in the system.

Better communication

No matter what type of development process you use, you're going to have to provide regular status reports to your user base or customer. Even if you're working on-site and you are in constant communication, you'll need to document your progress for a couple of reasons.

First, except when you're directly working with the president or owner of the company, those documents serve as a vehicle for your contacts to communicate to their bosses. If you don't provide status reports, your contacts will have to create them themselves, and

that's going to mean extra work for them, and it's possible that the information will not be reported accurately.

Second, you'll need documentation of your progress along the way for your own records—so you can track what you're actually doing, not what you think you're doing. Remember the adage, "That which gets measured gets accomplished." If you're not measuring what you're getting done, it's too easy to fool yourself about your progress. And many a developer has moved along a "great" relationship with a customer, thinking it's okay to skip progress reports, only to be brought up short all of a sudden by a suddenly unhappy customer who demands to see the results. Don't mistake a comfortable and easygoing working relationship as tacit permission to slack off on part of your job.

Finally, in order to determine your price, as you'll learn in Chapter 17, "Calculating Time and Cost for a Specification," you need to have collected a history of your costs, which is driven in part by these status reports.

The benefit of a RAD process is not that you can skip these reports—you still need to do them, but your users or customers will have a better sense of and confidence in where you're at.

More realistic expectations

Finally, as the users work with you during the development of the prototype, you'll have ongoing opportunities to set their expectations properly—and, if you remember, that's the name of the game in this business. They're far more likely to be happy with what they get because they'll have been asking for it all along. It's a lot more difficult for your kids or your spouse to complain about the results of a shopping trip if they accompanied you on the trip than if you went alone, right? Same thing here. It's more difficult for a customer to be unhappy about the functionality and operation of a system that they've had a hand in developing than if they feel you did it all by yourself.

They'll also get a better feel for how long it takes to accomplish your work, and as they see your progress, they can better gauge how many additional things they should ask for and how long it will take you to respond.

Disadvantages of RAD

There are disadvantages, too, though. One disadvantage is that it can be tempting to prototype well past the point of diminishing returns, and never get the job done. Once users see how easy it is to make a few little changes to the prototype, it's possible for them to request iteration after iteration, in search of the "perfect" application. At some point, you'll have to say "good enough" and postpone future requests for a subsequent version.

Another disadvantage is that it's easy for customers to perceive that you're nearly done with the app simply on the basis of a few prototype screens, and it can be a difficult challenge to convince them otherwise.

Agile Methodologies

"Agile Methodologies" is a loose term that refers to any number of a variety of ultra-rapid software development techniques. They all draw from the same pool of techniques and thought processes—pair programming, immediate turnaround of small modules, writing code to small test plans that were created before the coding started, and so on.

The first Agile Methodology (AM), and probably the most mature (all of three or four years old now) was "Extreme Programming" (XP). XP is a development style created and popularized by Kent Beck as the natural evolution to RAD in answer to the faster and faster turnaround cycles demanded by the "Internet generation" of software development—vague and ever-changing requirements combined with incessant pressure to deliver "yesterday."

Its core premise is that of starting a software project with a minimalist design that is put into production immediately, and that evolves constantly to add needed functionality. The development is carried out by a pair of programmers in front of one screen, with short turnaround times, who integrate and test the whole system several times a day, and write unit tests before the modules they are to be run on, and keep those tests running at all times.

Obviously, this is a radical departure from even advanced RAD techniques of rapid design and prototyping iterations, but given the history of software development—50 years and we're still in the Stone Age—XP has gained a number of adherents who are still looking for a better way to develop applications.

The advantage of XP is that the application is put into production very quickly—no waiting for six months or a year before the user sees anything concrete. As a result, this saves time and money. By definition, an application that is ready in a month will cost less than one that is ready in a year. Additionally, features that aren't absolutely necessary are delayed until "later," and thus don't produce a cost.

The disadvantage of XP is that it requires, er, extreme discipline. Many developers will attempt to mimic XP but only implement part of the process. This is like learning to swim with one arm and one leg "tied behind your back."

There is a contingent of developers who wonder if, generally, Agile Methodologies are more ideally suited to applications that do not "do data." If you think of some early Web sites where the database back end consisted of three tables, it's easy to see where this approach could work beautifully. But it's easy to wonder how effective this approach would be if you needed to have any significant data behind it. Are Web developers still working in this manner today? Could you realistically go about this incremental a design and have a data structure worth anything?

It's difficult enough to convert a well-designed data structure to a new schema and not have huge holes or major problems after you're done, given that the philosophy behind Agile Methodologies—write a bit of code, test a bit of code, install a bit of code—pretty much requires wholesale changes to data structures on an ongoing basis.

Agile Methodologies have only been around a few years, and so can still be considered "on probation," although that pronouncement will surely earn me nasty e-mails from avid supporters of one or more of the techniques. The jury is still out on whether these are going to be good for the long haul. Still, for certain types of projects, an Agile Methodology might just be the ticket.

Benefits of Agile Methodologies

All methodologies are intended to manage risk. Structured programming/Big Design is a stern reaction to CLHASWH (discussed later in this chapter) approaches, often termed "Cowboy Coding," and attempts to manage risk by concentrating on the analysis and design phases so as to reduce to a manageable certainty the actual construction of the software. Ideally, the uncertainty of construction can be reduced to the point where the client can be quoted a fixed

price and a firm delivery date by, some would say, nitpicking the details to death. In this extreme attempt to control and manage the process, the process takes precedence over delivering value, driving up costs and slowing the development process.

All software development has four major variables: resources (the people, money, and materials required), scope (what is the system expected to do when completed), time (how long it takes to complete the system), and quality (does the system do what it is expected to do).

In practice, quality is not really a variable subject to manipulation, as shortcuts that sacrifice quality for the sake of limiting costs (resources) or time-to-completion actually add to these other variables in the long term. As a result, one should regard quality as a fixed value in relation to the other three.

That said, you might consider that Agile Methodologies deal with quality in a different way than the other processes. The best definition of quality that I've ever run into has been simply "fitness for use." A rough-cast two-penny nail that you pick up at a hardware store may be of sufficient quality if you're building a doghouse in the backyard, but a finely crafted machine bolt may still not be of sufficient quality if you're using it on a space shuttle.

This "fitness for use" has two basic components: quality of design and quality of production. The design has to address the use issue, while the production has to meet the specifications of the design. A square wheel may be machined and produced perfectly to specifications (high quality), but its design (a device to enable a vehicle to travel over ground via an axle) is poor (low quality). An oblong wheel that was designed as a round wheel—the design was better quality but the production was poorer quality—would end up being a better solution.

Agile Methodologies can do a better job to produce the production of the solution, at the risk of a poorer design, because the system is being built in small pieces and thus the big picture may not be addressed until too late.

How Agile Methodologies deal with risk

Each of these variables represents certain dimensions of risk, and both structured programming and RAD fail, in some ways, to adequately manage those risks. Agile Methodologies, in general, attempt to address these issues by recognizing certain truths about the development process. As you read through this list, you should either be nodding your head knowingly, or underlining these points in bold:

- The user never has a clear idea of the requirements for the system, and determining these requirements is a process of discovery rather than documentation.
- The user will always require changes to the system during development.
- The user shouldn't be making technical decisions; developers should not make business decisions.
- The user is interested in features, not program modules.
- The user wants to see tangible results as soon as possible.

These truths create the following risks.

- The risk that the customer will spend more than they planned, wait longer than they want to, and end up with something that doesn't meet their needs.
- The risk that the developers will commit to a deadline or cost estimate on which they can't deliver.
- The risk that the developers will need to incorporate some new technology that they're uncertain of, and could jeopardize cost and delivery estimates.
- The risk that the customer will continually request changes to completed program code, delaying and/or significantly complicating the construction process.
- The risk that the overhead associated with the project will help the project feed upon itself, creating a monster of scope creep and turning the project into a death-march.
- The risk that non-technical managers will get involved in making technical decisions.

Agile Methodologies deal with these risks differently than other methodologies in that AMs were created specifically to deal with these risks head-on. Structured development was devised for the convenience of the programmer, not the customer. RAD moved along the route to an extent, and AM attempts to deal with software development methodologies to solve the customer's problem, not the programmer's problems. The others were created for the programmer's convenience. Here's how AMs deal with the risks, and attempt to compensate for the failures of other methodologies.

Risk 1: Cost overrun/extended time frame/failure to meet needs

Because the analysis and design phase of structured programming is highly uncertain (it's not unheard of for this phase to drag on for years), both the cost and time-to-completion for this phase are also uncertain. The risk is that the client will spend more than they planned, wait longer than they want to, and end up with something that *still* doesn't meet their needs. AM addresses this risk by providing something immediately, and at minimal cost, that explicitly meets the most important needs of the moment.

Risk 2: Failure to meet cost/time frame promises

With both structured development and RAD, at some point the developer quotes price and time delivery numbers, and then is committed to meet those. Given the great difficulty most development shops have in quoting reasonable numbers for these parameters (hence, this book!), it's highly likely that those numbers will not be met. AM addresses this risk by breaking down the deliverables into small entities where an overrun is less likely and doesn't have as significant an impact; reasons for overshoots can be incorporated into future deliveries in a continuous feedback loop, reducing the risk as the project progresses.

Risk 3: Incorporation of untested technology

The introduction of untested technology creates a brand-new risk: moving the system development from an engineering project to a research and development project. As a result,

cost quotes become cost estimates, and not very good ones at that. Delivery quotes are similarly out the window.

AMs address this risk in the same fashion as the aforementioned Risk 1 by breaking down the deliverables into small entities, where an overrun can be contained in a small environment so that it can be studied and the rest of the project can be rescoped and measured accordingly.

Risk 4: Change requests

The analysis and design phase can span so much time that the client's business model, supply and/or distribution chains, internal organization, market, and competitors can all change during this process. This further adds to the cost and time uncertainty of this part of the project. Once analysis and design has been completed and the project has moved to the construction phase, changes may (nay, will) continue to occur, and can negatively affect code quality. The risk is that the client will continually request changes to completed program code, delaying and/or significantly complicating the construction process. AMs address this risk, again, by providing something immediately, before requirements have had time to change.

Risk 5: Process focus creates a death-march

Structured programming/Big Design can become so mired in the process that the actual delivery of working program code that meets the client's needs becomes secondary, and the analysis and design and the documentation seen as integral to the process consumes so many resources that there isn't enough left to bring the process to completion. In addition, the process actually assists in the addition of new requirements, such that the project feeds upon itself, and the process ends up encouraging scope creep instead of reducing it. The process can be complicated by numerous repetitive forms to complete, and multiple sign-offs for every requirement and specification. The risk is that the project will turn into a death-march driven by unchecked scope creep.

AMs attempt to address this risk by minimizing overhead and process constraints, instead focusing on delivery of usable functionality in short order.

Risk 6: Span of control

One of the supposed benefits of structured programming is that it allows non-programmer project managers to control a process in which they can't actively participate. There is often distrust or anxiety on the part of the client, and this can extend to the analysts and project managers who don't really understand software construction, and thus eventually also begin to distrust the developers. Bureaucratic procedures are put in place in an attempt to control the propeller-heads in development, which further slows the process and increases costs.

AMs attempt to address this risk by creating small work units where the individuals best suited for each task actually perform those tasks.

Dealing with how long and how much

The major risk factor that Agile Methodologies do not address (but handle in a manner that's often acceptable to the customer) is the age-old questions "How long?" and "How much?"

Proponents of AMs argue that these questions cannot be answered—ever—until the customer can explicitly describe what it is that is going in "the shopping basket," any more than the shopping clerk at the grocery store can tally up your bill until he sees what's in the

basket. The attempt to create that explicit answer, given the ethereal nature of software development and the nature of the truths listed earlier in this section, will be met with failure until we have better tools and processes.

Proponents of AMs will argue that Agile Methodologies handle this failure in a manner that's often acceptable to the customer. Whether you agree, of course, is up to you, and depends on the situation you're involved in. The bottom line is that the customer has the final say in how well the answers to "How long?" and "How much?" need to be defined.

Comparison of Agile Methodologies with structured programming

Before proceeding with a description of how an Agile Methodology works, a discussion of the variety of AMs that exist is required. With a half-dozen books on the marketplace, Kent Beck's Extreme Programming is probably the most mature and most fully articulated of the Agile Methodologies. Others include, but are not limited to, Feature Driven Design, Adaptive Software Development, and Crystal Clear Methodology. This is not to say that XP should be the only methodology to employ. You may find that some of the practices of other Agile Methodologies represent a better fit for your shop, your clients, or your projects. However, because of the completeness of the XP model, its practices will more directly influence the remaining discussion.

Agile Methodologies differ from structured programming in one very significant way. Structured programming uses the waterfall model, and even RAD processes rely on a two-step process: specification, then development. Agile Methodologies use an iterative process from the very beginning—building a spec for a small piece of the application all the way through to delivering it, and then starting again with another spec for the next piece. The assumption of the waterfall model (as befits the engineering paradigm on which it's based) is that the development team is dealing with a static problem. However, software development is rarely dealing with a static problem.

A friend of mine once likened software development to changing the tires on a moving car. While all processes start with identification of the client's needs and a list of requirements, both structured programming and RAD processes then proceed to defining specifications and designing the entire system. Agile Methodologies, on the other hand, tend to formulate the requirements as a list of specific desired features, prioritize this list, and employ a cycle of specification, design, implementation, and delivery of each feature before proceeding to the next.

The goal of this approach is to accommodate (and even encourage) refinement, correction, and change of the requirements and specifications during the development process. As pointed out earlier, the customer's needs often change during development. Again, it's a normal and expected part of the process. The customer's needs also change as they're exposed to a working system; automation (or re-automation) of a process often changes the process, and gaps in the specifications and requirements often become apparent when the client has the opportunity to "test drive" a system.

The role of requirements and specifications

While the way in which it is expressed differs, defining the customer's need and collecting requirements is a central first step in both structured programming and an Agile Methodology.

This defines the scope of the project, and is the single most important tool for keeping the scope manageable within the constraints of the customer's budget and deadlines. Kent Beck has written that you can give your client the choice of two of the three "negotiable" variables (quality is non-negotiable): scope, cost, and time. The developers pick the third.

Defining the "wish list" scope, and then determining how much of this can be accomplished within the time or cost constraints, is the mechanism by which this most important variable is managed. If the customer wants to expand the scope, the developers can tell the client how this will affect cost or time-to-completion estimates. If the customer wants to limit cost, the developers can tell the customer what has to be trimmed from the scope. With an Agile Methodology, because the list of features is always prioritized (and re-prioritized as needed), it's usually clear that the items at the end of the list are the ones that need to go.

Specifications are also common to both approaches. This is what documents the business logic, the user interface, the work flow, and the data presentation and data collection rules for the system.

However, while both approaches rely on requirements and specifications, their role in the process is different. First, in an Agile Methodology, requirements documents and specifications are usually much less formal than in RAD processes, and certainly less than in structured programming. Their role is to facilitate communication, not control—the developers understand what the customer expects, and the customer understands what the developers are committing to deliver. Theoretically, requirements and specs could even be hand-written on index cards or the backs of envelopes. If forms are used, it is only to streamline the process, not to enforce rules about what blanks must be filled in.

Also, while requirements collection is a "first step" in the process in both structured development and Agile Methodologies, in Agile Methodologies it is revisited and manipulated at every opportunity—requirements collection is a process of discovery, not documentation. In an Agile Methodology, the requirements are used as the central mechanism for directing the development process, defining deliverables, and ensuring that the process delivers value.

As mentioned earlier, this process also continually tests how the customer values each feature. As new features are added to the list during development, the list must be re-prioritized. The choice of feature A vs. feature B is a binary decision. If added features require changes in the cost or time-to-completion estimates, or if other features must be postponed to another phase of the project or another fiscal year, the customer is forced to re-examine the value they place on each feature. Is adding feature A worth giving up feature B? Is adding a new feature worth delaying delivery by an estimated month, or adding $10,000 to the estimated cost?

To function as the central guiding force in an agile process, the requirements are prioritized, and the requirement at the top of the list is then fleshed out by the developers in cooperation with the client into detailed specifications, and the requirement is coded and delivered for the client's approval in a very short time period—usually 2-4 weeks. This process is repeated for each requirement on the list until the system is completed to the client's satisfaction.

This iterative process is the first of the two central components of an Agile Methodology, and is common to all of the Agile Methodologies with which we're familiar. The requirements can take different forms. XP uses the term "User Stories." Crystal Clear Methodology uses "Use Cases." Feature Driven Development uses a list of features. In all cases, the requirement unit is something that 1) defines a significant feature of value to the customer, and 2) is

something that can be implemented in a very short time period, usually 2-4 weeks. Note that there is no relation between features and what we as developers think of as modules. The difference is one of point of view. Modules are how we see the internal organization of the system; features are how the customer sees the system's functionality.

This distinction between features and modules is important. For instance, if you're working on a system that requires an order-entry component, as a developer you're likely to get sidetracked by all of the supporting modules that are required to maintain the data that the order entry component will require—customers, vendors, products, pricing, and so on. This can take weeks in itself, and when you finally get to the order-entry component, you might discover that the specifications for some of these supporting modules were insufficient or poorly understood—a one-to-many relationship between two data entities is discovered to be a many-to-many relationship, making one of the supporting maintenance modules incorrect. Thus, instead of getting sidetracked in this manner, you recognize that the customer is most interested in that order-entry component, and concentrate on getting that piece right, and only then are you free to concentrate on the supporting modules.

Because the requirements collection is defined as customer-defined features, all of which are more or less the same "size" in terms of difficulty to implement, it's often easier for the customer to prioritize all of these features. Recognizing that this can still present problems to some customer, XP places each feature (what they call a "User Story") on an index card, and the customer is asked to arrange them on a conference table in order of importance.

A friend of mine once had a customer who, when presented with a list of features on a whiteboard, would make a list of priority numbers next to each feature. They all had "1" written next to them, but some had "1+", "1++", or circles and underlines to indicate relative importance. In other words, the customer considered all of the features important, and resisted the attempt to prioritize. Index cards arranged on a conference table forces prioritization. Even though that particular customer would probably start stacking cards in the No. 1 position, their position in the stack still indicates a priority.

The assumption here is that the system will be constructed beginning with the most important features, and progressing first through the "must have" features, then through the "important but secondary" features, and finally through the "nice to have" features. This allows the customer to rest assured that even if the budget and time resources are consumed faster than anticipated, the system will at least meet their most pressing needs. The developers too get to play a role in this prioritization process, by moving up in the priority list those features that represent significant technical risk—usually those features that require use of new or otherwise unfamiliar technologies. This allows the developers the comfort of knowing that those unfamiliar and possibly difficult features won't pile up at the end of the process.

Estimating time and costs and billing

I mentioned earlier that XP and other Agile Methodologies don't do a very good job of estimating time and costs. This is true when you're considering the entire system. However, when you're looking at each individual module, it's a different story. Because part of the strength of Agile Methodologies is that they accommodate constant change, estimating these two important numbers is difficult when considering the entire project. On the other hand,

Agile Methodologies do allow low-risk commitments to delivery date and cost for each iteration. I say "low risk" because the commitments are to a small unit of the whole project. The cost of being wrong is thus proportionately less, and there is ample opportunity to improve during the project.

Requirements collection to develop the prioritized feature list for the entire system is a very open-ended process, and must be billed hourly. This provides an incentive to the customer to ensure that their employees do their part to keep this process moving. However, the developers share this burden, requiring them to manage this process and do their part to keep the client focused and on track to complete this phase of the project as efficiently as possible. As developers, we're out to deliver maximum value, and allowing a meeting to drift to discussion of the performance of the local sports teams or last night's sitcom episode while our clock is running does not fulfill this responsibility very well. It's tricky to get a client back on track once this thread drift occurs—the first few times. But as with anything, with practice you'll be able to get them marching again in short order.

Once this feature list is completed to everyone's satisfaction, regardless of what medium was used to document the process, the list becomes a deliverable and the client is billed for this work.

Similarly, the first step of each iteration is compiling the specifications or manufacturing instructions for the feature being implemented. Again, this is a smaller, but no less open-ended process, and must be billed hourly. However, once the manufacturing instructions are complete, you can determine how many people you are going to devote to this iteration, and commit to a fixed price and delivery date in writing. The client has a document that describes what you've committed to completing, and you have a recipe to use over the next 2-4 weeks to deliver it.

When the feature is completed, the latest build of the system is delivered to the customer for their review (along with the source code if that is part of your agreement), and it's accompanied by an invoice that details the hours spent compiling the manufacturing instructions, the cost for those hours, a description of the feature that was implemented, and the agreed-upon cost for that feature.

You might employ a "Specifications Document" that allows you to quickly wrap up a specification meeting and get to work on the actual construction of an iteration. This document can identify the participants in the specification meetings and the dates of the meetings, and will serve to inform the customer right away what they'll be billed for this work—both the hourly work to develop the specification, as well as the fixed cost for the finished feature when delivered.

Code Like Hell And See What Happens (CLHASWH)

Finally, migrating to the far end of the spectrum extends this trend of less design while getting your hands on the keyboard quicker and results in the "Code Like Hell And See What Happens" approach.

Every developer has heard the joke where the pointy-haired boss from "Dilbert" says to his team of programmers, "I'll find out what the users want. In the meantime, while you're waiting, you guys start coding." Unfortunately, like many stereotypes, this one has its roots too firmly planted in reality to be funny.

Sadly enough, and, unfortunately, encouraged by a lot of cowboy programmers, the first thing many companies expect to see upon engaging a software developer is some programs. Sounds reasonable enough, right? You hired a programmer, you should get some programs.

The advantage of CLH is immediate gratification—the engagement starts at 8 AM, and programs start to appear at 9:30 or 10 that morning. The disadvantage is that the programs produced are poorly designed, badly constructed, don't meet more than a superficial set of user requirements, and can't be maintained with less than a superhuman effort. By the time these problems are discovered, though, the CLH programmer is long gone, ready to wreak the same havoc on another unsuspecting customer.

Conclusion

Like the color spectrum, the choices of process available to the software developer range from the tightly structured process to the cowboy style of coding. They each have their place, but it's important to understand the pros and cons of each, and which type you're best suited for, for a particular application and set of customer requirements. And it's even more important to be able to communicate the resulting expectations to your customer.