METHODS & TOOLS

Global knowledge source for software development professionals **ISSN 1023-4918**

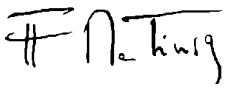
Winter 2002 (Volume 10 - number 4)

A Balanced Approach to Extremities

Three articles of this issue are dedicated to the eXtreme Programming (XP) method. This approach is linked to the Agile Software Development movement. In the recent years, methods like DSDM, SCRUM or XP have tried to solve the eternal dilemma between a structured approach and a people-driven process. They could be seen as evolutions of the RAD-like approaches of the past, where concept as small teams and timeboxing were already used. The wide spectrum of approaches to software development can be related to the diversity of the developer's population and the type of projects and environments.

If you look at the semantic opposition of the words used by concept like "Rational Unified Process" and "eXtreme Programming", the difference is obvious. The "extreme" name can explain the caution from established companies to adopt this approach (Agile Method is a more market-oriented denomination!)... and the interest of younger developers or small development companies. However, my opinion is that both approaches are not so different if you consider the people aspect. The days are over for "all-in-one" methods that imposed a unique path through a prescribed set of phases and tasks. Both approaches use the intelligence of their users to tailor the software development process to the knowledge of the domain, the environment or the type of software. XP offer a minimal set of principles and RUP is more like a big toolbox where you have to choose the right device for your work.

If many methods are now less rigid inside, there is still a lot of work to do in the software development community for an open view outside the methods. This is one of the point I like in the article of Pascal Van Cauwenberghe: the idea that an incremental software development approach is useful for some projects, but not for all. Unfortunately, companies that adopt processes tend to have the "one size fits it all" approach and they choose one method for all their projects.



Inside

Extreme Programming as Nested Conversations ..... page 2

Another Look at Incremental and Iterative Development ..... page 13

The Problems of Predictive Development ..... page 22

Getting the Most out of a J2EE platform ..... page 27

Extreme Programming as Nested Conversations

William C. Wake, William.Wake@acm.org
http://www.xp123.com.

Software is hard. It's hard to find out what's needed. The real requirements are hard to discover; plus, they change over time, and they change as a result of the software being created. But even once we know what's wanted, software is still hard. It's hard to master many details, and it's hard to merge together the results of a team's work.

Extreme Programming—XP—is a software development method designed to help teams create software more effectively. XP uses "simple rules" as a starting point for a team's process. XP's claim is that if we:

- put the whole team in a room together,
- force feedback through constant planning, integration, and release, and
- adopt a test-driven approach to programming then the team can be highly responsive and productive.

We'll look at XP as a series of nested conversations. You'll see a pattern repeated at each level: explore, plan, define a test, do the work, verify the result.

Nested Conversations in XP	
Month scale	Release planning, iterations, release to users
Week scale	Iteration planning, daily work, release to customer
Day scale	Standup meeting, paired programming, release to development team
Hour/Minute scale	Test, code, refactor

Three Voices

Let's consider three situations where you might develop a program.

First, suppose you have a problem, and the resources, time, ability, and desire to write a program (and that's the easiest thing to do). Then you may write the program to meet your needs. You may feel no need to separate roles.

Next, suppose you are a business owner and you have the resources and a need, but no ability or desire to write a program. You might hire a programmer to do it for you. It makes sense for you to distinguish your role from the programmer's.

Finally, imagine that you have the vision and need for a program, but you're not able to write it yourself, and you don't have the money or resources to hire a programmer. You might convince someone else to provide the money to hire a programmer. The person putting up the money will want to be involved (to be sure the money isn't wasted).

XP views teams as similar to this final scenario. An XP team has three sub-teams:

- Customer: Someone with needs and vision for a solution

- Programmer: Someone with the technical skill to create a solution
- Manager: Someone who provides an environment and resources for a customer and developer to work together.

The interaction between sub-teams is treated as a conversation between these roles. Underneath, each role is composed of several people, each with their own conflicts and compromises, but when they speak to the other roles, they speak with one voice.

Here, we see one of the ways XP simplifies reality. In reality, people have many skills: testing, analysis, programming, etc. But for the sake of the process, we act "as if" reality were simpler.

Key Points – Three Voices

- XP acts as if different sub-teams each speak with one voice.
- The manager provides the context for the team.
- The customer has a need to be addressed by software.
- The programmer implements the software.

The Context

Just as a choir needs rehearsal space and an accompanist, an XP team needs resources: workspace, computers, money, software, people, and so on. Management provides this context. They take responsibility for hiring, firing, space, tools, and so on. They control the flow of their investment; they may increase or decrease it.

XP asks for a special environment: a place where the whole team can sit together. Who is the whole team? It includes all the roles, but especially Customer and Programmers. Sit together means "really" together: in the same room (not in cubicles in the same building), where people can see each other. (XP is not saying a good software team can't be split across locations; it's saying that a team can be more productive if it sits together.)

People who are near each other talk to each other more. XP teams value communication and feedback, and collocation feeds both.

Key Points – The Context

- The whole team sits together.

Some Vocabulary

A *release* is the delivery of software, ready for use. Ideally, and usually, this is "use" by real end users, so the team gets feedback. A release is typically developed and delivered in one to three months, but some teams deliver every week or even more frequently.

An *iteration* is a time period during which the team develops software. Iterations have a fixed length, usually one or two weeks. Iterations are time-boxed: if a feature won't be done as planned, the scope of the iteration is adjusted—the iteration is not changed in length.

The Month-Scale Conversation

The whole project looks like this:
Release planning | Iteration | … | Iteration | Release

The conversation begins with an iteration or two of planning, and ends with a release to the end users.

To decide what will go into a release, the team will create a *release plan*. This plan shows, for each iteration, what features the system should have.

To create this plan, the customer will create *stories*—brief description of features, written on index cards. Here is a sample story card:

Create account
Enter desired username, password, & email address. Re-try if already in use.

The customer describes the story, but not every detail is necessary at this time. Rather, the conversation needs enough detail that the programmers are comfortable giving a ballpark estimate.

There will be a lot of conversation, but it won't be all talk for the week or two. The programmers have several jobs during this exploratory time:

- estimate how long each story will take to implement
- make quick experiments, known as *spikes*, to better inform the estimates programmers will assign
- set up the development and test environment
- create a super-simple skeleton version of the application, to demonstrate the pieces working together. (For example, a web-based course registration system might have one web page with a simple form of one field, sent to a back end that looks up one thing in the database.)

Most XP teams use a simple relative scale for the stories: 1, 2, 3, or "too big." (These numbers are often called *story points.*) It's helpful to have a rule of thumb as a starting point; you can use "one point = the team for a day" or "one point = a programmer for a week." But mostly, the stories will be compared against each other: "Is this one about as big as that one? Do we understand how to do it?" If a story is too big, the customer (not the programmers!) will divide it into two or more smaller stories until it can be estimated.

Using relative estimates is a little unusual, but it does have some benefits. It lets teams estimate stories by their "inherent" complexity, independently of the details of who will do exactly what part of the work. Furthermore, teams are often better able to start by saying, "these are about the same size" than to estimate a story's absolute cost. The team's ability to implement stories may change over time as a team learns and changes, but the relative costs of many stories will tend to stay the same.

The customer will write stories until they feel they've explored the possibilities enough to make a plan, and the programmers have estimated each story.

Given the estimated costs, the customer ranks the stories from most important to least. The programmers give an estimate of *velocity*: the number of story points they believe the team can implement per iteration.

The customer will arrange the stories into columns, with "velocity" points per column. The number of columns is determined by the number of weeks the customer (or management) wants to go until release. If all the stories don't fit, the customer can either defer them to a new release, or adjust the planned release date. (For most customers, it's better to drop features rather than slip a date.)

Here is a sample release plan, for a team with a velocity of four points/iteration:

Release Plan			
Iteration 1 List courses (1) Enroll in course (2) Summary report (1)	**Iteration 2** Create account (2) Login (1) Show schedule (1)	**Iteration 3** Drop course (1) Manage courses (3)	**Iteration 4** Fees (2) Logout (1) 1-sec. response (1)

This is a plan, not a commitment. It will change over time.

Until the release, the team engages in iterations. We'll look at the iteration-level conversations next.

Key Points – Month-Scale Conversation

- The customer describes stories.
- The programmers estimate stories (1 to 3 story points).
- The customer sorts stories by priority.
- The programmers estimate velocity (points/iteration).
- The customer creates a release plan.
- The team performs iterations, then releases.

Week-Scale Conversations: Iterations

A release plan is a high-level plan; it's not detailed enough for a team to work from directly. So, an iteration begins by creating an *iteration plan*. This is similar to a release plan, but at a finer level of detail.

The team first asks, "How many story points did we complete in the last iteration?" The team will plan for that many points for this iteration. This rule is known as *Yesterday's Weather*, on the theory that yesterday's weather is a pretty good predictor for today's. (For the first iteration, the team can use the velocity estimate from the release plan.)

The customer selects the stories for the iteration. The customer needn't select stories in the order used for the release plan; they can pick whichever stories have the most value given what they now know. They might even have new stories for the team to estimate.

The team then brainstorms *tasks*, to form an iteration plan. For each chosen story, the team lists the tasks that will implement it. One of the tasks should be "run customer test" (or the equivalent), as the customer's criteria determine when the story is done. Here is a sample iteration plan:

Iteration 2 Plan

Create Account (2)

- Account table with user, password, email
- Web page
- Servlet – check up acct, create it
- Run customer test

Login (1)

- Lookup account, create cookie
- Web page
- Run customer test

Show Schedule (1)

- Web page (result table)
- Servlet – lookup courses for logged-in user
- Run customer test

Individuals then sign up for tasks. Some teams assign all tasks at once, others sign up as the iteration progresses. In either case, you should be able to look at the chart and know who's doing what.

Note that "Run customer test" is a task for each story. One of the jobs of the customer is to specify a test for each story. Ideally, the tests are ready even before iteration planning, but they need to be done before a "completed" story can be counted. Ron Jeffries describes requirements in XP as having three parts, "CCC" –Cards, Conversations, and Confirmation. Customer tests provide the confirmation. (Recall that "customer" may be a team including professional testers. The customer need not implement the test, but should specify and own it.)

Once we have a plan, the conversation shifts down to daily activities. At the end of the iteration, the team will deliver the application to the customer for final assessment of the iteration's work.

Key Points – Week-Scale Conversation

- Use Yesterday's Weather to determine the velocity to use.
- The customer selects stories adding up to "velocity" points.
- The team brainstorms tasks.
- The team signs up for tasks.
- The team does daily activities, and delivers the result of the iteration.

Day-Scale Conversations: Daily Activities

The team begins its day with a stand-up meeting. (This practice was adapted from the Scrum process.) The team stands in a circle, and each person takes a minute to tell what they did yesterday, what they plan to do today, and what's in their way. (If they need help, they can ask for a followup meeting.)

About halfway or two-thirds through the iteration, the team should do what Ward Cunningham calls a "Sanity Check": "Are we on track? Will we finish the stories we planned? Do we need more stories?" If the team is behind, they may need to re-negotiate the iteration plan with the customer: defer a story, choose a simpler one, etc. (These adjustments – dropping or adding stories – are what allow velocity to change from iteration to iteration.)

After the standup meeting, the team will break into *pairs* – groups of two. XP specifies that production code will be implemented by *pair programming*, for immediate design and code review. (See *Pair Programming Illuminated*.)

The pair selects a task, and works on it together. Periodically, they *integrate* their work into the main body of code, or pick up what others have integrated. After a couple hours, the pairs may swap around.

At the end of the day, the pair will integrate one last time. If they've completed a task, they'll mark it off. (Very rarely, they may decide they're on a wrong track, and abandon the last bit of work.)

Several things enable such flexible teamwork:

1. The team has daily standup meetings and sits together, so everybody has some idea of what everybody else is up to.

2. The team owns the code jointly; any pair can change any code it needs to.
3. Each pair integrates its work into the mainline many times per day.
4. The mainline is kept working: no code is checked in that causes a regression in the tests.

Key Points – Day-Scale Conversation

- The day starts with a standup meeting.
- Pairs form and shuffle during the day.
- Code is integrated into the mainline many times per day.
- The mainline is kept in a working state.

Hour- and Minute-Scale Conversations: Programming

The programmers talk to each other, and the customer, as they work on their task.

There is also a "conversation" with the code. XP uses a development style known as test-driven development. You'll hear many different words and phrases associated with this technique: *simple design*, *test-first programming*, *refactoring*, *green-bar/red-bar*.

To create a new feature, the programmer writes a new, small, "programmer" test for part of the feature. The test fails, of course, since the feature doesn't exist yet. So the programmer implements the feature in a simple way to make the test pass. Then they refactor (systematically improve the design) to remove any duplication and leave the code communicating as well as it can. The whole cycle repeats until the feature is completely added. Each trip through the cycle takes a few minutes.

At the end of a session, the program will have a new feature, and an automated set of programmer tests that demonstrate it. The code and tests go into the mainline, and the team will keep both the feature and the tests working while other code is added.

Programmers have found that this style yields programs with decoupled and encapsulated designs, and the code is known to be testable (as it has been demonstrated).

Key Points – Hour/Minute-Scale Conversation

- The programmers and customers talk to each other during the day.
- Programmers have a conversation with the code: test, code, refactor.
- This creates new features and automated programmer tests.

Beyond the Mechanics

The description above represents a starting point for a team's process. Real teams will evolve their process over time.

Teams have found that there is a synergy in the XP practices. They support each other, so you can't just pick and choose without finding some new practices to balance the missing ones.

Many teams have found that adding a regular retrospective is helpful: it builds in time to reflect on how things are going and how they can be improved.

Conceptual Frameworks

We've looked at XP from the "mechanical" side; now we'll consider its underpinnings from some other perspectives:

- Values and practices
- Agile methods
- Self-organization at the team level
- Empirical vs. defined processes
- Emergence at the code level
- Lean Manufacturing

Values and Practices

The first XP book, *Extreme Programming Explained* (by Kent Beck), introduced a framework of "values" and "practices" for describing XP. Values are more fundamental; practices are activities or skills that are compatible with the values, and form a starting configuration of team skills.

The values:

- Communication
- Feedback
- Simplicity
- Courage

The practices:

On-site customer	Testing	Pair Programming
Planning Game	Continuous Integration	Simple Design
Metaphor	Collective Ownership	Refactoring
Short Releases	Forty-Hour Week	Coding Standards

(Most of these practices were worked into the description of the mechanics described in the first part of this paper. There are other lists of practices that use different words to explain the same themes.)

Agile Methods

Extreme Programming is an example of what are known as *agile methods.* Some others include:

- Scrum (www.controlchaos.com): This is probably the most philosophically compatible with XP. Scrum uses one-month iterations, and its own approach to planning. It allows for large projects via a "Scrum of Scrums."
- Crystal Clear (www.crystalmethodologies.org): "Management by milestones and risk lists." Crystal Clear is the simplest in a family of methods.
- FDD (www.featuredrivendevelopment.com): Plan, design, and build by feature, in a model-driven approach supported by a chief programmer.
- DSDM (www.dsdm.org): Model and implement through time-bound iterations. (DSDM is an outgrowth of earlier RAD approaches.)

See www.agilemanifesto.org for a manifesto and principles statement from a number of leaders in agile methods, and www.agilealliance.com for the home of the Agile Alliance.

Self-Organization of the Team

Among agile methods, XP and Scrum stand out as relying on a team to organize itself. This flows from the *team* taking on responsibilities. It's also due to the lack of built-in role specialization. XP teams value specialized skills; but they don't pigeonhole people into having only one aspect. (Database programmers who want to learn some GUI programming can pair with someone who has more experience.)

The team takes responsibility: the team accepts stories, and the team finds a way to do them. In most XP teams, individuals accept tasks. Even so, they're understood to have the full support of the team. If they ever need help, they ask, and it will be given.

The physical environment encourages self-organization too. When people sit together and eat together, they build bonds and realize, "We're in this together," and "What affects you affects me."

Defined and Empirical Processes

(Scrum brings this vocabulary into play as well.) Consider making cookies. You have a recipe, and you follow it. If you make another batch, with the same ingredients, in the same proportion, in the same oven, you expect to get the same result. This is an example of a *defined process.*

Consider instead the process of creating a cookie *recipe*. The value comes from its originality. You might try a number of variations, to get just the right result. Iteration is inherently part of the process; this is an *empirical process*. (Reinertsen, *Managing the Design Factory*, suggests the cooking example.)

Software development tends to be an empirical process: the goal is not to get the same result a team got earlier, but to create something new. Experimentation is a critical part of this, not a failure.

In spite of the concrete description in the first half of this article, XP is in the "empirical" camp. It accepts that there will be experimentation on all levels, including experiments about the process itself.

Emergence at the Code Level

One of the unexpected aspects of XP is its flipping around the development cycle from "analyze-design-code-test" to "analyze-test-code-design" (Ralph Johnson). One way to design is to speculate on the full design that will be needed. Another approach is to intertwine design and development. XP follows the latter approach: build a little something, then evolve and generalize the code to reflect the design.

Martin Fowler's book *Refactoring* catalogs "code smells" (indicators of design problems) and "refactorings" (safe transformations that can address problems). These provide (usually) local improvements to code.

The team also looks for more global improvements. Pair swapping and shared ownership mean that people will be exposed to more areas of the code, so able to spot similarities among disparate sections. The team's search for a metaphor (shared understanding of the system) can help this too.

Why is this emergence? Because simple rules (smells and transformations) lead to something perhaps unexpected: globally good design.

Lean Manufacturing

The automobile industry has moved from assembly lines to lean manufacturing. Traditional assembly lines "push product" as fast as possible; inventory is regarded as an asset. In lean approaches, a product is "pulled" from the system, and inventory is regarded as a source of waste.

XP's approach to planning and implementation strives for "just in time" work.

- At first, stories described with just enough detail to allow an estimate.
- For each iteration, just enough stories are expanded with tests and details so the stories can be broken into tasks.
- For the current task, then the pair will write a test and implement just enough code to make the test pass.
- For the resulting code, the pair will refactor to reflect the design as it's now understood.

Suppose a team is doing 100 stories, at the rate of ten per week. This might be the schedule for an XP team:

- Week 1: 100 stories described and estimated
- Week 2: 10 stories get customer tests; those same 10 stories get unit-tested, coded, and refactored.
- …
- Week 11: The last 10 stories get customer tests, unit-tested, coded, and refactored.

Because the team is completes the highest-value stories first, the earliest iterations are the most valuable.

Compare this to a strict waterfall:

- 100 stories get analyzed
- 100 stories get designed
- 100 stories get coded and unit-tested
- 100 stories get tested

At some level, there's the same amount of total work (though my bet would be on the first team). But look at it from a flow perspective: we don't see any results until stories come out of testing.

Think of a story as inventory. When it has been analyzed, designed, and coded, but not tested or deployed, it has a substantial investment at risk. The XP pipeline lowers the risk: a story gets everything done at once, and spends less time at risk.

Conclusion

XP challenges traditional software development processes in several ways: everything from how a team is structured to how code is implemented comes in for scrutiny. The XP practices represent an effective way to help a team learn what software is needed and develop that software, while respecting and valuing each person on the team.

Further Reading

- The XP series from Addison-Wesley: *Extreme Programming Explained* (Kent Beck) is the first in the series; *Extreme Programming Explored* is my contribution; *Testing Extreme Programming* (Lisa Crispin and Tip House) is the latest addition.
- The Agile Software series, also from Addison-Wesley: *Agile Software Development* (Alistair Cockburn) is a good starting point.
- *Managing the Design Factory: The Product Developer's Toolkit* by Donald Reinertsen
- *Pair Programming Illuminated*, by Laurie Williams and Robert Kessler
- XP web sites: www.extremeprogramming.org, www.xp123.com, and www.xprogramming.com
- "XP on One Page" is a mini-poster available at http://www.xp123.com/xplor/xp0202/index.shtml

Going Round and Round and Getting Nowhere eXtremely Fast? Another Look at Incremental and Iterative Development.

Pascal Van Cauwenberghe, pvc@nayima.be

'T was brillig, and the slithy toves Did gyre and gimble in the wabe
From Jabberwocky
"Through The Looking Glass" – Lewis Carrol

Introduction

Surprisingly many people still wonder what's wrong with the safe and predictable sequential phased approach, also known as "waterfall". This despite the fact that since many years, a lot has been written about iterative and incremental development. Still, most people involved in software could not give you a clear definition of what those two words mean to them. When they are able to give a definition, they often contradict each other.

Into this confused world, the proponents of "Agile Development" and especially Extreme Programming proclaim that they go beyond "traditional" iterative and incremental development methods. What are we to do? How do we approach software projects? How do we reach our targets reliably, quickly and efficiently? When do we gyre, when do we gimble?

Defining our terms

"When I use a word", Humpty Dumpty said in a rather scornful tone, "it means just what I choose it to mean – neither more nor less".
"Through The Looking Glass" – Lewis Carrol

We must first choose what we want our terms to mean.

A phase = a period in which some clearly defined task is performed. Just imagine we require the following phases: Analysis, Design, Coding, Testing and Integration & Deployment. Each phase has some inputs and some outputs.

To iterate = to perform some task in multiple passes. Each pass improves the result, until the result is "finished". Also called "rework".

An iteration = a period in which a number of predefined tasks are performed. The results are evaluated to feed back to the next iteration.

An increment = a period after which some part of functionality is completed to production quality level.

A release = some coherent set of completed functionalities that is useful and useable to the intended users of the system.

With these definitions we can have a look at different ways to organize a software project.

Different ways to plan a project

The waterfall – straight to the target

The **sequential**, **phased** approach, better known as "Waterfall" is the simplest and potentially fastest approach: we analyze the problem, design the solution, implement the code, test the code (are we allowed fixing the coding bugs?), integrate if necessary and deploy. Done.

The attraction of this process model is its simplicity: you do each of the steps and then you're done. If you want to know how long it takes, just add up the time required for each phase. These times are just estimates, so your total is also an estimate. But it makes following up on the project's schedule easier. Once, for example, analysis and design are done, they are done.

The simplicity of this model attracts two kinds of people:

- Remote stakeholders like upper level managers or end-users who have no need to know the exact process. They really just want to know when the outputs will be delivered.
- Teachers and educators who need to present a simplified model of a process as a first, simplified introduction to the subject. Unfortunately, it is this first approximation that sticks in the minds. Or do students skip all the following lectures?

The spiral – iterate to incorporate feedback

Of course, it's never as simple as the waterfall process suggests: we do make errors during analysis and design. When Walker Royce[i] originally described his process, there were feedback loops back to analysis and design. They were quickly "simplified" away. So, this was the original **phased, iterative** approach:

- Perform analysis, until you think you are ready
- Design, until you think you are ready. Should you find any problems in the analysis, feed them back to the analysis. Perform another iteration of the analysis, to improve it.
- Code the design, until ready. Should you find any problems in the design, feed them back. Perform another iteration of the design, to improve it.
- Test the code. Should you find any problems, start another iteration of the coding.
- Integrate and deploy. Feed back up any problems you encounter.

In short: you go sequentially through the phases. Some phases have more than one iteration, if we see that the output of that phase is not perfect. In more than one sense, the waterfall is de optimist's interpretation of the spiral model.

As Boehm[ii] noted, the later a defect is caught, the higher the cost to fix it. We are therefore encouraged to only proceed cautiously to the next phase if the previous phase is relatively complete. As software engineers have noted, the likelihood of finding defects becomes higher as we come to the later phases. This is because each phase delivers a more precise, more "testable" output that the previous phases.

There is one management problem with the spiral model: how do you schedule the iterations? How many iterations will be needed? This makes estimating and tracking the project's progress more difficult and unpredictable.

Iterative development – getting better all the time

Where the spiral process views feedback and iteration as exceptional events, iterative development assumes we will make mistakes, we will need feedback, we will need several iterations before we get it right.

In **iterative, phased** development we design our project plan around the idea of iterations and fit the phases in them. For example, we can plan to have five iterations:

- In the first iteration we will do 90% analysis, 10% design
- In the second iteration we will do 30% analysis, 50% design, 20% coding
- In the third iteration we will do 10% analysis, 30% design, 60% coding
- In the fourth iteration we will do 10% design, 50% coding, 40% testing and bug fixing
- In the fifth iteration we will do 50% testing and bug fixing, 30% integration, 20% deployment.

This process takes into account the reality that we almost never get anything completely right the first time. At first, we will get a lot of feedback. Gradually, with each iteration, the output from each phase stabilizes and we need to do less rework.

Iterative development with incremental delivery - growing software

One complaint we could have with the previous process, is that it takes so long to see the result of the work. Indeed, the software isn't ready until the absolute end of the project. What if we need the results faster? We could use **iterative development with incremental delivery**, a variation on the previous process.

The basic idea is to release the software several times to its users, each time with more functionality. Each release is complete, useable and useful to its users. Each release adds more functionality, preferably the most important functionality first. How do we schedule our project?

- First, we go through several iterations of analysis and design, until we are pretty sure these phases are mostly complete. We can then select the content and schedule of the increments to be developed. Let's say we have identified three increments A, B and C, to be delivered in this order to the users.
- For increment A, we iterate through all required phases, until the software can be released. We expect to rework the analysis and design very little.
- For increment B, we iterate through all required phases, until the software can be released. We expect almost no analysis or design work.
- For increment C, we iterate through all required phases, until the software can be released. We expect almost no analysis or design work.

We can picture this process as four applications of the above process. The amount of analysis and (architectural) design decreases in each step. The process delivers functionality after the second, third and fourth steps.

How do we select the contents of the increments? Architecture-driven processes would select the elements that have the most impact on the architecture (and thus the design). Risk-driven processes would select the most risky elements. Users would prefer the most useful or urgent functionalities.

Agile software development – Extremely incremental

Agile software methods like "Extreme Programming[iii]" take another approach, based on the following assumptions:

- You should welcome feedback and rework, in stead of trying to avoid it
- You should deliver software to your users in small increments and deliver functionalities in the order that the users want.

The development process is then scheduled as follows:

- Decide on the length of incremental releases. All increments are the same length. Decide, if necessary, which ones will be "internal releases", only used internally for evaluation and feedback, and "external releases", which are released to the end-users.
- Gather an initial list of requirements. New requirements can be added at any time.

- Before the start of each increment: users (or their representative) prioritize the requirements and assign the requirements to releases. As many releases as required can be planned, but typically the contents of all releases, except the one being implemented, are subject to change.
- During the increment, the requirements are analyzed and broken down into small micro-increments. These increments are small enough to be fully implemented in a day or less.
- To implement a micro-increment, the developers analyze, design code, test, integrate and deploy iteratively. They iterate very quickly, so as to get as much as possible concrete feedback to guide their work.
- Each increment has a fixed length. If needed, scope is reduced (by the user) to meet the target date.

What's so extreme in "Extreme Programming"?

What's the difference between Extreme Programming style planning and iterative development with incremental delivery, such as is possible with the well-known and accepted "Unified Process" framework?

The assumption that **you can do architecture and design incrementally**. Indeed, XP spends very little time upfront to define an architecture and overall design. Instead, we immediately start with the first increment to deliver functionality. All architecture and design work is done to satisfy the requirements of the current increment. XP does design in the following ways:

- A *System Metaphor* describes an overall architecture and vocabulary that all participants understand and agree with. The Metaphor is refined iteratively over the whole project.
- *Test-First* programming designs the code incrementally by defining executable tests that the program unit must comply with.
- *Refactoring* reworks the design to iteratively adapt the design to changing knowledge, environment or requirements.
- Whiteboard design sessions, CRC card sessions… and other group design techniques.

The even more controversial assumption that **you can do analysis incrementally**. At the start of each increment, we only examine the requirements that have been scheduled for this increment. XP does analysis in the following ways:

- Brief requirements, called *Stories*, are elaborated interactively between the development team and the story author.
- Requirements are formalized into executable "*Acceptance*" tests, which specify and verify the compliance of the software with the requirement.
- Requirements are allocated to releases using the *Planning Game*, where developers and customers optimize delivered value versus development cost.

The assumption that **you can deliver software features incrementally, in whatever order yields most benefit for the customer**. If the software can be delivered incrementally, the users will get the benefit of the functionality sooner and thus get a faster return on their investment. As the customers can choose the delivery order, they receive the most important and valuable features first. The use of these increments generates lots of valuable feedback to drive the following increments.

What's the best way to plan a project?

The short answer: it depends. It depends on your project, your team, and your environment.

If the assumptions of XP hold, we can gain a lot of **benefits** from working incrementally:

- Early delivery of useful software.
- Delivery of functionality based on the value users give to that functionality (business value driven instead of architecture or risk driven).
- Clear, tangible feedback on progress: each increment delivers *finished, production quality* functionalities.
- Clear and regular feedback on the quality and fit of the software from the users to the development team.
- The ability to adapt the project plan.
- Clear scheduling and predictable delivery
- The development of the simplest possible system that satisfies the requirements, without any unnecessary adornments and complexity.

What are the **dangers** of this approach?

- By only looking at small pieces (increments) of the software, we may have to do massive rework, which *might* have been avoided if we analyzed or designed the whole system correctly.
- We might "paint ourselves in a corner": get a system which is incapable of supporting some new requirement.
- We may be unable to add "global" requirements, such as globalization, performance, and security… A typical example is trying to retrofit "security" to an application that was not designed with this requirement in mind.
- We might go slower because we have to restart analysis and design sessions for each increment, instead of doing it once at the start of the project.

So, it's not so easy to define how we should organize our projects.

A few heuristics

If your requirements are volatile or incomplete, if the environment changes or if you wish to be able to anticipate to future events, work incrementally. If the requirements and environment are stable, you can optimize your process by looking at all the requirements upfront. When we use an investment analogy, we can compare agile scheduling with options, where we pay a small investment to be able to postpone our buying/selling decisions, versus shares, where we make the full investment upfront. The potential gains from shares are higher (because we don't pay the price of the option), but the risk is also higher.

If you are not familiar with the domain, haven't built any similar architecture, work incrementally. Use the feedback to design your system and to avoid unnecessary complexity. If you've done lots of similar systems, you can do more upfront. Still, you can always use the feedback to keep your system simple.

If you can't deliver incrementally, don't deliver incrementally. But, even if you can't deliver incrementally to your final user, you might deliver incrementally to internal users, such as QA, product managers… And even if you can't do that, deliver incrementally within your team. That way you can still get the benefit of early feedback.

If you know a requirement is going to be more difficult to work on if you only tackle it later, work on it now. This can happen with "global" properties of software systems like localization, scalability, and security. You can still get the benefit of business value driven scheduling for most requirements if you take those few exceptions into account: "Did you want security with that, Sir? With security, this feature will cost you X. Without, it will cost you Y. If you wish to add security later, it will cost you Z." Iterative (re) design techniques like refactoring and the emphasis on simplicity does help to keep your software "soft", more malleable, more accepting of changes.

If your team is big, do enough work upfront with a small team so that you can decompose the work on the system on the basis of a reasonable (but not perfect or final) architecture.

My approach

I prefer to err on the side of doing too little work upfront, rather than too much. That's because I've learned some things by applying Agile processes:

- If you use simple designs and keep them simple by refactoring you can keep your software so changeable that you can accept most requirements without much trouble. On a well-functioning XP team it feels as if you could go on forever adding features, if only the users could keep coming up with them.
- If you are afraid to change something, because you fear the refactoring will be a lot of work and break things, do it often. For example, in my team we've learned to do database refactorings often, updating our schema every few days, with a minimum of work and without our users noticing.
- No matter how well you know the domain or the architecture, you will learn new insights from feedback. The most important thing I've learned is to write code that is a lot simpler than it used to be.
- Customers are uneasy with the apparently complex or chaotic scheduling of iterative and incremental processes. They much prefer the safety and simplicity of a waterfall process. If you can involve them and have intermediate incremental deliveries, within the overall "waterfall", they quickly appreciate the feedback and steering power that you give them. And they will never work any other way again.
- Keep your team as small as possible. A smaller the team requires less communication overhead, less upfront planning and decomposition and gives more agility to respond to changes.
- The assumptions of XP might not hold completely, but often it's close enough. For example we can assume that most requirements can be analyzed, planned and implemented relatively independently so that users are free to schedule them as they see fit. We can handle those exceptions, where there are dependencies or risks, appropriately, so that we have the best possible schedule from both a functional and technical standpoint.

Conclusion

Incremental software development, with short increments, is a very useful technique to reduce project risk and to deliver value faster to users. Each release provides opportunities for useful feedback and replanning, so that the final product meets the users' real requirements. The project is at its most "agile" when we consider only one increment at a time.

We must balance this with some forward-looking actions, to avoid some pitfalls and to go faster. We can plan and implement several increments or perform some upfront analysis and design before starting the increments.

When shouldn't we work completely incrementally?

- When we have a lot of knowledge and experience in the domain, the architecture and the environment. We can "invest" our knowledge to enable us to take the "best road" to our destination.
- We think the cost of implementing the requirement will go up sharply as the project progresses.

There are a few heuristics and you should discover more. You must choose the right combinations, based on your experience, your environment, and your task. And you must always be ready to learn from feedback and to adapt your plan and your process accordingly.

---

[i] See http://www.sei.cmu.edu/cbs/spiral2000/february2000/Royce/ for an overview of RUP as an example of a "Spiral process"
[ii] "Software Engineering Economics" – Barry Boehm – Prentice Hall 1981
[iii] "Extreme Programming Explained" – Kent Beck – Addison-Wesly 2000

The Problems of Predictive Development (or, Why Fixed Price Contracts are a Bad Idea)

Steve Hayes, steve@khatovartech.com
Khatovar Technology, www.khatovartech.com

Predictive development

Most current software development is based on a predictive model. The customer predicts requirements, and the development team predicts a plan (including cost and delivery date). Once the project is started the plan is considered to be "correct" and deviations from this plan are treated as errors.

This is a great approach if you have complete knowledge of your starting point (the current business), you can analyse your requirements, and you can confidently project all this knowledge into the future, taking into account changes in your business, changes in the market, and changing technology. If you can do all this (and more) then you can come up with an exact plan for your project, including exact costs and exact delivery dates. Most people can't do this, especially at a detailed level.

How do predictive processes fail?

Predictive processes fail in simple ways. The software is produced on-time and on-budget, but no one wants to use it because important features are missing (the requirements were incomplete) or wrong (the requirements were inaccurate). The project is reported as on-track until shortly before delivery, when new information comes to light that requires major rework. Or the project gradually slips further and further behind schedule because of changes to the specification (often presented as "clarifications"), or because the developers simply overestimated their productivity. Sometimes the business requirements change dramatically mid-way through the project, often before any working software has been delivered.

Why do we do this?

We do things this way because our historical model for software development is an engineering project. In an engineering project (say, building a bridge) it's very expensive to change something once construction has started. Historically this was also true for software, and many software development processes reinforce this tendency by adding expensive change control processes.

SO if we had an environment and process that kept the cost of changing software low, would that make predictive development obsolete? Not necessarily. If a project has a large number of conflicting stakeholders, and these stakeholders need to reach agreement before there are any changes to the software, then predictive development is probably still our best bet. We need to reduce both the customer cost of change and the software cost of change before we confidently change our development approach.

Is customer cost of change the barrier?

Some people argue that the customer is the party insisting on complete, detailed requirements before a project is started. This may be true, but we need to ask ourselves why customers take this approach.

One reason may be that it's truly expensive for them to change the requirements, perhaps because they have a large number of stakeholders. But it's also possible that the customer is being reactive. Having heard from developers that software is very expensive to change, the customer has decided that they need to keep costs low by fixing detailed requirements. Having expended a lot of effort to produce precise requirements, the customer wants a precise project plan. To produce a precise project plan, the developers ask for more detailed requirements, and so it goes. When the software developer is external, this situation usually results in a fixed-price contract.

Who benefits from a fixed-price contract?

The short answer is, usually no one. As we've seen, it's very difficult to write a complete, accurate requirements statement. There's almost always negotiation between the developers and the customers during the course of the project, though it's usually called "clarification".

For any given feature, it's in the customer's interest to argue for slightly more work, and it's in the developer's interest to ask for slightly less work. This leads to an adversarial relationship between the parties, a relationship that's not conducive to creating effective software.

What's the alternative?

Many analyses of predictive software development note that most requirements statements actually lead to increased costs (in one study quoted by James Martin, over 80% of the costs of defects have their roots in requirements based errors). The conclusion is usually that more time needs to be spent getting requirements correct, but that's not the **only** possible approach. Another approach is to acknowledge, openly and explicitly, that the requirements statement is only a rough guideline, that the first project plan is at best a guess (by both the customer and the developer), and that change will be an explicit part of the project.

To make this feasible you need to have a process that helps you keep the cost of change low. That's where Extreme Programming (XP) comes in.

Extreme Programming

Extreme Programming (XP) is an agile development methodology that has become increasingly popular over the last few years, particularly in the USA and Europe. The core values of XP are communication, simplicity, feedback and courage. XP also includes twelve key practices that direct day-to-day software development. Each practice contributes to keeping the cost of software change low, and to leveraging this low cost of change. Rather than go into details of all the XP practices, this paper will focus on how XP projects address project planning and changing requirements.

XP treats software development as an exploration, with the customer and the programmers working together to address immediate problems, install working software, and use the lessons from this experience to direct the next piece of development. The customer steers the XP

development team so that they are delivering the maximum possible business benefit at any point in time.

The beginning of an XP project resembles the beginning of a predictive project. The customer determines some feature set and discusses this with the programmers. The critical difference between the two approaches is the level of detail. A predictive project needs lots of details. An XP project needs just enough detail for the programmers to put a rough estimate on the feature, and in particular to get an idea of the relative sizes of the different features.

The underlying assumption is that these features are just a list of things that **may** be developed, and expending effort detailing something that never gets developed, or is developed in an unanticipated way, is a waste of resources. The estimates are in terms of **ideal** engineering days - days where there are no interruptions, no software or hardware glitches, and the implementation moves smoothly. Of course, no one has days like this.

Next, the XP team estimates how many ideal engineering days they will get through in a week. This is the team's **velocity**. A predictive project makes a similar estimate, perhaps just mentally, when it determines the project plan. The difference here is that the XP team acknowledges that this is just a **guess**. If the team has worked together before on similar projects then it might be a good guess, but it's still a guess. It needs to be validated. A predictive project never gets the chance to validate its velocity - it commits to a fixed delivery date before they do any development.

The first guess of velocity, along with the feature estimates and the business estimates of benefits, is enough to determine if the project looks economically feasible, but everyone involved will want to validate the estimates as soon as possible. And since one of the core values of XP is feedback, it's good if the validation is repeated throughout the project.

Validation of both development velocity and business benefits is embedded in the structure of XP. An XP project is delivered in short iterations (two- or three-week iterations are common, and some projects use one-week iterations). At the beginning of the iteration, the customer picks the features that will deliver the most business benefit, bearing in mind the estimates against the features. They have a dialogue with the programmers, adding the detail that's needed to perform the implementation for that iteration.

At the end of the iteration the programmers deliver **working software**. The programmers record how many ideal engineering days of work they delivered in the iteration, and use this to reestimate the overall schedule. The customer uses the working software to validate requirements and business benefits. Based on this, the customer chooses the features for the next iteration (which may be completely new), and the cycle continues.

The most important point here is that the programmers deliver working software to the customer every few weeks. XP also contains practices to ensure that the quality of this software is high and that there is very little, if any, regression from one release to another. This is important because no one wants the customer to waste time finding bugs in the release - everyone wants the customer to spend time **using** the release, and giving the programmers feedback to direct the next release. This is exactly what's happening on hundreds of XP projects right now.

How does XP treat project risks?

If the customer has excellent predictive powers then the releases could look a lot like the milestones in a predictive project, and we haven't really gained much. But what happens if we run into some of the risks related to predictive projects?

If the programmers are less productive then they anticipated, we find out very quickly, because delivering working software doesn't leave much room for uncertainty. Knowing that fewer features will be delivered lets the customer focus the development on the most important ones. If the changes are so drastic that the project's business case is compromised, the project can be cancelled. And the customer gets to make these decisions every few weeks.

Working software can also be used by the final customers, and we can do this every few weeks. Getting regular feedback from the final customers dramatically increases chances that the final software will be accepted and used.

Changes of direction are no longer subject to acrimonious negotiation - they're expected at the end of every iteration.

What does this mean for fixed price contracts?

A fixed price contract isn't the best arrangement for a XP project. Instead, the customer and the developers should commit to begin development based on the overall schedule, and decide at the end of each iteration whether or not they should continue. This may sound onerous, but it's easy to make "continue" the default decision, and only do something formal when the direction of the project looks dubious.

Customers sometimes object that this arrangement gives them less control, because there's no guarantee they'll get **all** the features delivered. But this has always been true. There are many fixed price contracts where the developers failed to deliver, and this wasn't clear until the end of the project. Some of these projects have even ended up in court (for example, the RAVC vs Unisys in Victoria, as described in "The Age", September 18, 2001). Other projects go to completion on time, delivering functionality that's no longer relevant. Aside from development costs there are also opportunity costs - if a project is going to fail you want to know as soon as possible. Forcing regular deliveries of working software is the best way to find out. So the customer actually has **more** control of an XP project, not less.

Developers sometimes object that this arrangement gives them less control, because they can't predict the end of the project and this makes resource planning harder. But let's look at how a project might finish early. It might finish early because the business value has been delivered far quicker than expected, and the customer is very happy. It might finish early because the development has gone well, and has revealed that the business benefits aren't what were expected. The customer may not be thrilled by this, but better to discover it early in the project rather than later.

Making the customer happy is the basis of a long-term relationship, and generally leads to more business. On the other hand, the project might finish early because the programmers can't meet the expectations they (or the salespeople) set. This is certainly to the detriment of developers who oversell, but not to the detriment of developers setting reasonable expectations.

Conclusion

In the modern, turbulent business world it can be very difficult to make accurate predictions about anything. Unfortunately, many businesses use software development practices that assume detailed knowledge of the future, and often this leads to unsatisfying software development experiences. Software development methodologies like XP embrace change, rather than resisting it, and thrive on uncertainty. Many businesses that currently use fixed price contracts for software development should instead consider iterative developments and financial arrangements based on XP practices.

Bibliography

- Beck, K., "Extreme Programming Explained: Embrace Change", Addison Wesley Longman, 1999
- Beck, K., and Fowler, M., "Planning Extreme Programming", Addison Wesley Longman, 2001

Getting the Most out of a J2EE platform

Habib GUERGACHI – CTO, Groupe SQLI

Most major enterprises have made their choice of application server, and many have even launched important projects based on this new infrastructure for executing transactional, secure business objects. The great majority of these enterprises have opted for the J2EE standard, with all the advantages and risks brought by adopting a new platform as a strategic component in an information system.

Our experience in major transactional projects means that we can now take a step back and look at vendors' sales pitches and the received marketing ideas. So how do we make the most of the functional and technical richness and potential of J2EE, without taking the slightest technological risk?

To answer this, we will need to start by explaining how J2EE technology is developed, looking at its complexity, highlighting those modules that have attained maturity and those that haven't. We shall then give our opinion on which parts of J2EE to use, and which to lose in mid-2002.

JCP ensures the functional richness and durability of Java

Sun, the inventor of Java, totally controls the Java™ brand and remains a very influential player when it comes to managing this industrial standard, offering protection against any attempted "hijack" by Microsoft.

In order to protect its progeny, Sun created and organized the "Java Community Process" (JCP), which takes the form of an international group governed by an "Executive Committee" (EC) in which influential players such as IBM, BEA and CISCO are represented – the first two being particularly powerful promoters and decision-makers.

The JCP organization ensures the richness and durability of the Java standard. The standardization process is kept democratic and transparent through the Java Specification Requests (JSRs). A member of the JCP can, at any time, submit a request to update of a particular aspect of Java. To do this, the requesting party must formalize and submit a JSR application, clearly explaining the objectives and methodology of the request. The JSR then enters a completely transparent, responsive review circuit.

Power and influence struggles do occur, but they do not have any major impact on the standard itself. On the one hand, the JCP organization makes sure that Java remains independent from the influence of powerful players, and on the other, Sun does not miss any occasion to demonstrate the force of its lethal weapon, which it has already tested with success against the enemy forces of Microsoft!

J2EE, richer and more complex then you might think

J2EE (Java 2 Enterprise Edition) standardizes transactional architecture based on the Java language. Vendors implement the J2EE standard and market it through J2EE application servers.

Put simply, J2EE standardizes the JDBC JSP/Servlets technology on the one hand, and EJBs on the other (see table at the end of this article: "J2EE 1.3 specifications"). The EJB standard specifies, on the basis of a single technical structure, different types of EJB components designed to carry out different functional roles within a transactional J2EE architecture.

Within the EJB standard we find both Session Beans and Entity Beans. A session bean layer is generally used for managing an application session, and monitoring the navigation of an identified user. The session bean layer then hands over to the entity bean layer which implements business objects. These handle execution of management rules, consistency of transactions, security of execution of methods, database persistence, and so on.

Session beans can be "stateful" or "stateless". This means that they either do or don't manage a functional state (such as a list of parameters) between the different stages of the user's session. Entity beans may feature Container Managed Persistence (CMP entity beans) or Bean Managed Persistence (BMP entity beans). CMP beans bring complete transparency to persistence of business objects in a database, whilst BMP beans require specific development of this persistence layer (in JDBC for example) or need to call on an external technology to ensure persistence.

J2EE, what to lose in 2002

What is reassuring about a J2EE platform is that when a standard is simple and above all, adopted unanimously, it can be implemented in a completely standard fashion across all application servers.

The clearest example of this is JSP/Servlets/JDBC. This standard is perfectly well implemented by all market vendors. It is therefore quite possible to port JSP/Servlets/JDBC applications between market application servers.

The clearest counter-example is found in CMP entity beans. This standard is very young, and is certainly not unanimously adopted. The result is poor implementation, even with the most advanced, powerful application server vendors in the sector. What is more, the unanimity of CMP entity beans is contested by another standard, which is technically much simpler and yet functionally equivalent – Java Data Objects (JDO).

CMP entity beans still pose risks, in technical, performance and even durability terms, given the availability of other solutions such as JDO. Indeed, one of their main advantages – the standardization of automatic persistence mechanisms – still has functional and even structural weaknesses, and lacks maturity in implementations.

If it is absolutely necessary to implement Java applications that observe the object paradigm from end to end, we would recommend using proprietary business object persistence technologies from specialized vendors such as Versant or TopLink (Oracle).

J2EE, what to use right now

Clearly, The J2EE standard is structuring the technological foundations that will underpin the information systems of major firms in the future. The lessons that have been drawn from experiences in major transactional projects, now in production, are starting to take root.

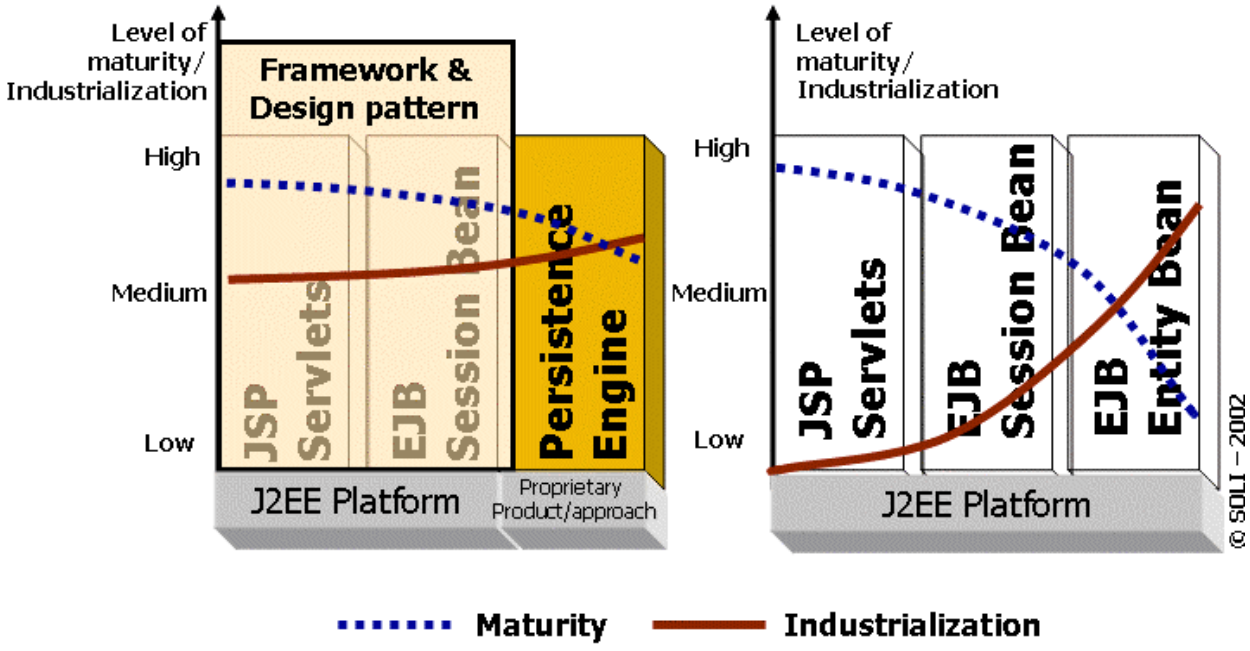
The JSP/Servlets/JDBC has proved its merits in a number of major projects. It offers an

excellent standard of performance and, above all, ensures portability between application servers. However, it does not offer all the features necessary to industrialize production and operation of applications in an enterprise. This drawback can be overcome by using Java frameworks such as Struts or by implementing Design Patterns such as DAO (Data Access Object) for persistence.

EJBs generally prove useful when it comes to standardizing and industrializing the techniques for developing and utilizing software components, but they also bring added complexity, cumbersome implementation, and are not without shortcomings in terms of maturity and performances.

Session beans and BMP entity beans may severely affect performances, but they do feature sophisticated characteristics that make them essential for implementing applications based on software components. By making considerable effort to acquire technical, design and development expertise, the risks of using session beans and BMP entity beans can be limited, and their functional richness exploited.

Opting for pragmatism: JSP/Servlets/JDBC + Frameworks



The options for J2EE adoption

Companies starting out with Java and wishing to begin J2EE development without taking risks in terms of technical aspects, durability, development scalability, or operability of applications, should certainly envisage intensive use of EJBs in later phases, and rely on JSP/Servlets/JDBC technology.

For enterprises with greater experience in Java technologies, we would recommend methodically applying strict standards, carrying out quality checks at all stages of the project, opting for training in software design, J2EE architectures, design and programming techniques, and so forth.

Note: J2EE 1.3 Specifications

API	Comments
JDBC 2.0 (Java DataBase Connectivity)	API for connection to relational databases
RMI/IIOP 1.0 (Remote Method Invocation / Internet InterOrb Protocol)	Remote component invocation mechanism
EJB 2.0 (Enterprise JavaBeans)	Server component model
Servlet 2.3	Web services extension API (equivalent to CGI)
JSP 1.2 (Java Server Pages)	Server-side scripting API (equivalent to ASP or PHP)
JMS 1.0 (Java Message Services)	MOM connection API
JNDI 1.2 (Java Naming and Directory Interface)	Directory connection API
JTA 1.0.1 (Java Transaction Services)	Transaction management API
JavaMail 1.1	Mail sending API
JAF 1.0 (Java Activation Framework)	API for composition of complex emails
JAAS 1.0 (Java Authorisation & authentication services)	Security service management API
JCA 1.0 (J2EE Connector Architecture)	API for connection to legacy systems

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig; e-mail franco@martinig.ch ISSN 1023-4918
Free subscription: www.methodsandtools.com