But Will It Work For Me?

Kathy Iberle
Hewlett-Packard
18110 SE 34th Street
Vancouver, WA 98683
Kathy_Iberle@hp.com
www.kiberle.com

Abstract

This article defines software engineering "practice cultures" as groups of people who hold shared assumptions about their business model and situation. These shared assumptions drive their choices of software engineering practices, often in ways that are not obvious. The author visits a number of practice cultures and explores the relationship between the software engineering practices favored by that culture and situational factors such as project size, cost of fixing post-release errors, regulation (or lack of), and economic drivers. An understanding of that relationship is important when evaluating whether a practice will work in a new environment.

Biography

Kathy Iberle is a senior software test engineer at Hewlett-Packard. Over the past 17 years, she has been involved in software development and testing for products ranging from data management systems for medical test results to inkjet printer drivers and, most recently, e-services. Kathy has worked extensively on researching appropriate development and test methodologies for different situations, training test staff, and developing processes and templates for effective and efficient software testing.

Copyright Kathleen A. Iberle, 2002

This paper was written for and presented at the 2002 Pacific Northwest Software Quality Conference.

Acknowledgements

My thanks to the participants of the Software Test Managers Roundtable (STMR), whose stories from a wide range of software development environments contributed substantially to this article. Facilities and other support for STMR are provided by Software Quality Engineering, which hosts these meetings in conjunction with the STAR conferences.

Special thanks to the many reviewers who provided informative and thought-provoking comments: James Bach, Bernie Berger, Bob Hollister, Barney Levie, Jonathan Morris, Carol Peterman, Bill Tuccio, and the reviewers from the Software Architecture Group at University of Illinois in Champagne-Urbana: Ralph Johnson, Brian Marick, John Brant, Bill Davis, and Weerasak Witthawaskul.

Why Don't We All Do the Same Thing?

The libraries and bookstores are overflowing with papers and books on software engineering practices. There's thirty years or more of experience behind us – so why haven't we discovered the one right way of doing software? Why do we still use different practices?

Because we're different! Far too often, I hear this when someone simply doesn't want to use the practice in question, or is convinced that she won't be allowed to use it. But in truth, we really are different.

In any flock of software engineers, different groupings can be identified. The members of a group share some common assumptions – assumptions about the business, assumptions about goals, assumptions about users, assumptions about risk. A group of practitioners who share a set of assumptions can be said to belong to the same *practice culture*, even if they don't know each other.[1] People working in the same culture tend to read the same journals, use the same language, attend the same conferences, and use the same practices. Some individuals are completely unaware of the existence of any other practice cultures, and others are completely convinced of the innate superiority of their own practice culture. I often see people coalesce into cultural groupings during lunch at large conferences – the aerospace and medical folks cluster together, happily comparing test plan formats and discussing reliability measurements, while the shrinkwrap software crowd is in another corner talking about agile methods and rapid testing.

Amazingly enough in this imperfect world, the different software engineering practice cultures actually work pretty well. Over time, practice cultures develop methodologies and practices that fit their situation (Bach 1999). They also tend to develop mythologies to explain the "rightness" of their practices. As with other human cultures, there is a strong tendency to believe that all people should recognize the superiority of one's culture and immediately rush to join it. There is also a tendency to believe that everyone already belongs to one's culture, and to be puzzled and hurt when members of another culture do not behave in the time-honored traditions of yours.

Books, journals, magazines, and methodologies are based on implicit, often unspoken assumptions about the situation experienced by the authors. The stated practices usually do work in those situations – but the question remains as to whether the practices will transfer successfully to your situation. If you identify the originating practice culture and understand its assumptions, you can then evaluate whether the assumptions on which the practice is based are likely to be true in your situation also. Recognizing your own practice culture and its differences from other practice cultures will help you:

- learn from other practice cultures
- work with members of other practice cultures
- evaluate new practices for effectiveness in your own situation
- move into a new job in another practice culture

[1] This concept was identified as a "community of practice" by James Bach , (Bach 1999). The term has also been adopted by the knowledge management community with a somewhat different meaning, so I have chosen to use the term "practice culture" in this article, as suggested by Mr. Bach.

Assumptions Behind the Practices

A practice culture is the collective creation of a group of people. It represents the group's common beliefs and practices. Most individuals live in one practice culture and have learned the beliefs and practices of that particular culture. Individuals who move between cultures generally carry their original culture with them and acculturate gradually, often with some degree of "culture shock" during which the practices of the new culture seem very foreign and unreasonable. I would expect that individuals such as contractors who switch back and forth frequently between cultures don't change their practices with every new job, but instead either stick with their original practices or create a cross-cultural amalgam of practices.

When looking at a practice culture, there are several things to consider:

- What assumptions does this practice culture make?
- How do those assumptions lead to "best practices" – that is, practices that are favored within that practice culture?
- Are there "best practices" that don't appear to follow logically from the assumptions? This suggests that there are assumptions unknown to you.
- Do any of the assumptions involve factors that have changed in the last few years?

The assumptions behind a practice are difficult to find because they are usually not listed neatly and explicitly with the practice description. Sometimes assumptions are stated as beliefs or even as incontrovertible truths, rather than assumptions. Other assumptions are left unspoken, because they are so obvious to the people in that culture. This is a trap for the unwary - even smart people from other cultures can miss the point completely.

For instance, if an inch or two of snowfall is predicted, many inhabitants of Seattle plan to stay home for the day. Coming from Michigan, I thought this behavior unnecessarily timid if not downright silly – until it actually did snow. The next morning, I skated downhill for less than a block before deciding to preserve my car and myself by prudently remaining at home. Two of my assumptions about driving in snow had turned out to be incorrect – Seattle has a lot of hills and the roads are not salted. Even though I had walked up and down those hills for months and noticed the lack of rust on the numerous older cars, it hadn't occurred to me that together this would add up to dangerously slick roads.

Assumptions typically are about things other than technology – for instance, James Bach (Bach 1999) lists capabilities, goals, and situation. A discussion of when a practice will work (or not work) usually names at least one and often several of the author's assumptions. For instance, Highsmith discusses a couple of classification methods for criticality when discussing agile methods, saying that regulatory requirements may preclude the use of some agile methods for life-critical software. (Highsmith 2002).

The Major Practice Cultures

So what are the major practice cultures? There are a number of different ways to divide the software engineering community into cultures. A variety of factors for being different are cited by the authors or critics of various practices – project size, customer type, risks incurred, number of regulations that affect the product, and so forth. For this paper, I chose to sort first by *business model* – that is, how do people profit from writing this software? This didn't result in clear groups with no overlaps (as you'll see later) but I am using this division for several reasons:

- The way in which money flows and what profit depends upon appears to either drive or constrain quite a few of the other factors.
- The effects of the business model on practices are frequently overlooked completely by practitioners
- I didn't see a better sorting rule that accounted for all the differences I could see.

Sorting by business model is not completely unprecedented - James Bach's list of practice cultures from his article "Good Practice Hunting" (Bach 1999) is very similar.

Business Model

The primary business models currently appear to be:

- **Custom systems written on contract:** We make money by selling our services to write other people's software.
- **Custom systems written in-house:** We make software to make our own business run better.
- **Commercial products:** We make money by writing and selling software to other businesses.
- **Mass-market software:** We make money by writing and selling software to consumer.s
- **Commercial and mass-market firmware:** We make money by selling objects which happen to require embedded software.
- **Open-Source:** We make software for our personal satisfaction. (This doesn't include all open-source software - some is written in a for-profit environment and later "freed", and some comes from academic projects.)
- **Academic:** We make software to use in our research – in biology, chemistry, physics, psychology, economics, and many other fields as well as computer science.
- **Internet**: There are multiple business models using Internet technology now, and a single practice culture has not yet formed. More on this later….

The underlying economic assumptions in different practice cultures are rarely stated in articles or books about practices. When I switched practice cultures myself, it took a while to realize that the reasons behind the practice differences very often traced back to economics. My new colleagues would mention the economic factors in passing, but often didn't explain the connections between the economics and the practices.

There are, of course, ways to make money and reasons to write software that don't fit neatly into one of the above categories. If a particular combination grows significantly, eventually there will be a critical mass and we see one (or more) new practice cultures arise. For instance, the mass-market practice culture didn't yet exist in 1980, whereas the custom systems practice culture certainly did. Existing practice cultures also grow and change over time, such that a description of the custom systems practice culture today will not be the same as it was in 1980.

Situational Factors

Each business model has a set of attributes or situational factors associated with it. Assumptions about situational factors are more likely than the business model to be stated in a process or practice description. Situational factors are often derived from or limited by the business model – both the way the money flows, and the end purpose of the software itself. For instance, software meant for consumer purchase is unlikely to be life-critical.

Here's a list of situational factors that seem to influence the choice of software engineering practices:

- **Criticality:** The potential for harming the user's or purchaser's interests varies depending on the type of product. Some software can kill you when it fails, other software can lose large sums of many people's money, and yet other software can do nothing worse than waste the user's time.
- **Uncertainty of Users' Wants and Needs**: The requirements for software that implements a known business process (such as the U.S. tax code) are necessarily better known than the requirements for a consumer product that is so new the end users don't even know that they want it.
- **Range of Environments:** Software written to use in a specific company needs to be compatible only with that company's officially supported computing environments, whereas software sold to the mass market has to work with a wide range of environments. The skill sets of users in the mass market also vary more widely than those of users in a specific company.
- **Cost of Fixing Errors:** Distributing firmware fixes is a lot more expensive than patching a single website.
- **Regulation:** Regulatory agencies and contract terms can require practices that might not otherwise be adopted, or may set up a situation in which certain practices become useful. Some situations require process audits, which verify that a certain process was followed to make the product. (The term "audit" is also used in some fields for an activity which verifies that the end product works as intended, which is not the same as a process audit.)

- **Project Size:** Multi-year projects with hundreds of developers are common in some businesses, whereas shorter single-team projects are more typical in other businesses.
- **Communication:** There are a number of factors in addition to project size that can increase the amount of person-to-person communication needed, or make accurate communication more difficult. Some of the factors seem to show up more frequently in certain practice cultures, whereas others appear to be randomly distributed.
    - **Concurrent Developer-Developer Communication:** Communication with other people on the same project is affected by the way the work is distributed. In some organizations, senior staff design the software and junior staff do the actual coding and unit testing (as opposed to having the same individual design, code, and unit test a given component). This practice increases the amount of developer-developer communication needed. Specialization can also cause communication difficulties when taken to the point where specialists start assuming knowledge on the part of others or are themselves missing a piece of the picture.
    - **Forward Developer-Developer Communication:** Maintenance and enhancements require communication with developers forward in time. This is easiest when the developers tend to stick around, and thus the communication is with yourself.
    - **Developer-Management Communication**: Project status needs to be reported upwards, but the amount and form of communication that managers believe they need varies rather considerably.
- **Organizational Culture:** The organization itself will have a culture that defines how people operate. Highsmith (Highsmith 2002) summarizes work by Moore[2] that defines four organizational cultures:
    - **Control –** "Control cultures, like IBM and GE, are motivated by the need for power and security."
    - **Competence –** "A competence culture is driven by the need for achievement: Microsoft is an obvious example."
    - **Collaboration –** "Collaboration cultures, epitomized by Hewlett-Packard, are driven by a need for affiliation."
    - **Cultivation** – "A cultivation culture motivates by self-actualization … and can be illustrated by Silicon Valley start-up companies."

  Organizational culture affects communication practices quite strongly, possibly more strongly than the practice cultures themselves. For instance, a friend of mine who was an in-house software designer for a control-culture auto manufacturer complained one day of the time she spent explaining the difference between 0 and O to one of the "coders" – a person with a few months of training in COBOL and no other background. My reaction "this is weird" probably stems from many years in a collaboration culture, rather than from differences between the in-house software business and the commercial software business. Organizational culture can also drive a practice culture to adopt or cling to practices that are not necessarily a good fit for the rest of the situation - Highsmith discusses the problems of adopting agile practices within a control culture. (Highsmith 2002).

It's important to keep in mind that people's beliefs about the situational factors of their practice cultures are assumptions, which may or may not match the current facts. It's very hard for any one person to have enough data to establish facts. Most assumptions are generalizations of currently true situations, but some represent situations that used to be true but have changed. Assumptions can have a lot of staying power, especially if they get into widely read literature.

It surprises some people to find more than one practice culture within a single company, yet this appears to be not uncommon in companies with a broad variety of businesses. Hewlett-Packard, with its history of autonomous business units, has developed distinctly different practice cultures in commercial software and mass-market software to suit the situational differences between these areas. Microsoft's work, while not covering such a broad range as Hewlett-Packard's, still shows some differences in practice: "Applications like Word or Excel, or consumer products like Works or Complete Baseball, tend to have shorter schedules, smaller teams, and more precise estimates of delivery dates than do system products such as operating systems…. The vision statement and specification document for systems products are more complete and detailed earlier in the lifecycle … Systems products also tend to have longer testing periods." (Cusumano 1995)

[2] The cultural models are based on work in Schneider (1994).

A Visit to Some Practice Cultures

Let's take a look at some of the major practice cultures in the United States. For each one, we'll first look at the business model in more detail, and then explore the situational factors. But wait – are these situational factors fact or fiction? How far can you trust me?

The statements about the situational factors are based on one or more of these sources:

- My own personal experience within Hewlett-Packard: this covers commercial software (mechanical engineering instruments – now part of Agilent; medical products – now part of Siemens; business inkjet printers), mass-market software (consumer inkjet printers), and both commercial and mass-market e-services.
- The experiences of people within Hewlett-Packard with whom I worked closely for an extended period of time: this covers all of the above plus both commercial and mass-market firmware.
- Stories gathered from conference presentations and books by people from a variety of practice cultures.
- Conversations with people from a variety of cultures, many of which occurred at the Software Test Managers Roundtable meetings.
- Stories from the various people who reviewed this paper.

The practice cultures in which I've had personal experience are probably more accurately described, particularly since I've been comparing notes with other practitioners in the same fields for some years. The ones in which I haven't had experience are probably less accurate. With that caveat in mind, let's go exploring…

Custom Systems Written on Contract

Business Model

An entity decides to hire another company to write a custom software system, or to customize and install a system provided by that second company or purchased from a third company. A contract is drawn up to capture the agreement for delivering the system. The profit for the company providing the services comes from staying within the allotted budget and avoiding penalties for late delivery. Normally, the contract requires a detailed set of specifications for the work to be done. Once the customer has signed off on those specifications, the contractor can charge extra for every change request thereafter.

Typical software written or substantially customized under contract for a specific customer includes banking networks, insurance software, billing and payroll systems, and a wide variety of military applications such as missile guidance systems.

Contract software is a very big business in the U.S. The two largest and longest-running purchasers of custom software appear to be finance companies and the U.S. military. Finance and military people have been using computers since the 1940s, and actively discussing what we would today call "software engineering" practices since at least the 1970s, if not earlier. In fact, the very earliest software was all custom-written for a specific use, so this business model has a great deal of history behind it. The history, coupled with the large number of people involved, has resulted in a huge amount of software engineering literature. A substantive portion of this literature is written by companies such as IBM that have historically provided both one-to-a-customer systems and customizable systems meant for more than one customer.

Software that was written under contract for the U.S. Department of Defense (DoD) ended up in this category along with software written under other types of contracts because I didn't find major differences in either practices or assumptions between software written for the DoD and for the rest of the contract world – in fact, it was clear that the other contractors had adopted many practices that originated in DoD work. It does appear that the DoD contracts are more detailed and prescriptive than the non-DoD contracts, but that didn't drive radically different practices as far as I could tell. However, I didn't get much information about DoD work from actual practitioners, so I may be missing something important in lumping DoD work in with other contract work.

The Situation

- **Criticality:** Software failures in finance systems can seriously damage the purchaser's business interests. Software failures in military systems can be life-critical, although quite a bit of software purchased by the military is actually business software for which the consequences of failure are similar to the consequences of failures in finance systems.
- **Uncertainty of Users' Wants and Needs:** Since the purchasers and users are an identifiable group of people, they can be located to find out what they want. In general, they do have a pretty detailed idea of what they want – nobody is interested in your creative re-interpretation of their payroll system. On the other hand, the process to be implemented isn't always well-documented, the users may disagree on steps of the process, their requests may assume technology that doesn't exist, business needs change during the project, and sometimes people just plain change their minds. Given all this, it's hard to understand why the literature in this practice culture is so emphatic about the need to establish all requirements up front – it may be that the contracting process itself forces people to believe that requirements should not change, although conversely the contracting process might be based on experiences from a time when requirements didn't change as much.
- **Range of Environments:** The purchasing organization has usually decreed a small set of target environments to hold down their costs, will offer or require user training if needed, and the users themselves are not such a varied lot as the general populace. This results in a range of environments that is both clearly defined and relatively narrow compared to the other practice cultures.
- **Cost of Fixing Errors:** Generally there are reasonably priced ways to distribute fixes – much of the software will be on servers in an identifiable building, and the location of client software is usually tracked.
- **Regulation:** Software for the Department of Defense must be written in accordance with a huge list of regulations, most concerning the process of producing the software. Finance software is not subject to regulation in the same way, although contracts vary. Not infrequently the contract requires process audits to prove that the organization followed its stated processes. The customer also often expects regular status checks.
- **Project Size:** Big seems to be the operative word here. Really big. Many dozens of people over two years seems merely average, whereas the biggest have hundreds of people over multiple years. There is some data suggesting that smaller projects are really much more prevalent than larger projects (Highsmith 2002), but the literature definitely reflects larger projects. This may be an example where the situation has changed but the assumptions about it have not, or it may be that funding for software engineering research is concentrated on large projects, and the publications reflect that.
- **Communication:** The practice of splitting design and coding between senior and junior staff shows up sometimes in this practice culture. Since the systems and projects are large, there are often separate people or even separate departments for analysis, database design, and so forth. In addition, maintenance contracts can go to people other than the original authors, so both concurrent and forward communications can be complicated. The companies themselves are often large, whether or not the projects themselves are large, which adds extra layers of status reporting.
- **Organizational Culture:** The companies that write software on contract often have a control culture. This seems logical since many of them have ties to the military.

These situational factors and capabilities together lead logically to a set of assumptions.

- **Delivery on time and within budget is very, very important**.
- **Software that reliably delivers correct output is very, very important.**
- **The requirements can and should be known in detail up front**.
- **Projects will be large and communication paths complex.**
- **We must be able to prove that we did what we promised.**
- **We need plans and regular status reports flowing upwards.**

Favorite Practices

Now let's explore how some practices popular in this practice culture make sense, given the situational factors and the assumptions.

Lots of documentation. Documentation is valuable as a communication medium when *project sizes are large*. Written documentation is often more effective than whiteboard or hallway chat when *the communications paths are complex*, which happens when people are widely separated by geography or job. In addition, some documentation is often required to *prove that we did what we promised*, and *requirements known in detail up front* also implies documentation of those requirements.

Capability Maturity Model (CMM). The CMM is a product of the Software Engineering Institute (SEI), which is funded by the U.S. Department of Defense (DoD). The SEI's stated mission is "to provide the technical leadership to advance the practice of software engineering so the DoD can acquire and sustain its software-intensive systems with predictable and improved cost, schedule, and quality". (SEI 2002) The CMM emphasis on careful estimation and project and detailed project management is completely in line with the assumption that one must be *on time and within budget* and with the assumption that *plans and regular status reports* are necessary. Large companies and *large projects* are both assumed by much of the CMM literature, such as Watts Humphrey's classic Managing the Software Process. (Humphrey 1989)

The CMM is also very popular with contractors in other industries because, just like defense contractors, they also need to be *on time and within budget*. In addition, CMM certification has become a selling point for many software contractors, so CMM practices may be adopted in order to get certification.

Waterfall Lifecycle. The waterfall lifecycle was originally invented to give *large projects* enough structure to be able to plan and steer towards *on-time delivery*. Newer "iterative" methods, such as Staged Delivery, plan in smaller chunks, which generally fulfills the need for planning while allowing more flexibility. However, it is easier to coordinate multiple *large projects* using the simpler structure of a waterfall lifecycle. Sometimes you find that the official high-level schedule appears to be a waterfall lifecycle for coordination purposes, and the actual activity on the team level is more iterative.

Formal Requirements and Analysis, Unified Modeling Language, Rational Unified Process: These practices stem from a combination of the complexity of the software, the need for *correct output*, and the practice of having multiple departments implementing different areas resulting in *complex communication paths*. Sophisticated diagramming processes to aid communication make sense here.

Process Audits. Extensive auditing procedures make sense when you must be able to *prove that you did what you promised*.

Rapid Application Development (RAD) and Joint Application Development (JAD). These "rapid" practices were developed as an alternative to the time-consuming practice of getting users and developers to agree on *requirements in detail up front* via an exchange of written documents, with all the concomitant misunderstandings and rework. These practices are consistent with the assumption that the requirements can and should be known up front – they are simply a different approach to getting there.

Cleanroom: Cleanroom is a process pioneered at IBM and supported and used by the Department of Defense and many of its contractors. The aim of cleanroom is "the development of high quality software with certified reliability" (SET 2000), leading to *reliable delivery of correct output*.

Custom Systems Written In-House

Business Model

In-house software is written when a need or opportunity to improve the bottom line of the company can be demonstrated. There are a variety of ways to fund projects – the larger projects require bigger decisions so they are

funded in a more formal way with a need for up-front estimates in detail, whereas smaller projects may not require as much justification. There are rewards for staying within the allotted departmental budget, but overruns are not fatal in the same way that they are in the contractual world. There are some projects that have to meet a deadline (like Y2K), and some projects that can be put on hold if the department runs out of money. Projects may be split up into sub-projects and approved separately over a period of time.

The typical software written in-house includes such things as order-tracking and online timecards.

The Situation

- **Criticality:** Most of the software is essential to the company's business purposes, although there will be occasional experimental projects of less criticality. In an organization whose business is something other than software, the criticality of the software may be underestimated by the people setting the budget.
- **Uncertainty of Users' Wants and Needs**: Many of the projects done in-house are very similar to those contracted out, so logically the level of uncertainty should be much the same as that of the previous practice culture. However, the discussion above pointed out that the traditional assumption that requirements should be stable may be based on economic factors (i.e. the way a contract works) rather than actual stability. Whether or not that is true, the "agile" practitioners in this practice culture are vocally asserting that the requirements in fact are uncertain and do change rapidly. The in-house developers have the opportunity to dramatically reduce uncertainty by talking directly to the end users, an opportunity often not available to developers in other cultures.
- **Range of Environments:** Again, the range of environments is limited by the company's own rules about its people's environments.
- **Cost of Fixing Errors:** The responsible department generally has easy access to the software, although distribution of fixes to a large number of clients can be problematic. If the functionality is centralized on a small number of servers, it can be quite inexpensive to release fixes.
- **Regulation:** Unlike the software provided on contract, generally neither regulatory issues nor process audits play a big role.
- **Project Size:** The size ranges from small to quite substantial.
- **Communication Between Developers:** As far as I have seen, the same people generally do design and coding. However, specialized staff for analysis, database design, and so forth seems typical in this business. Status reporting is of particular importance in this culture because the project budgets are often "overhead" in the larger company picture, and so projects often need to be repeatedly justified.
- **Organizational Culture:** Varies.

Again, the situational factors and capabilities together lead to a set of assumptions that are very similar to those of the contract world, with a few key differences (underlined):

- **Delivery within budget is very, very important, <u>but delivery on time may be less so.</u>**
- **Software that reliably delivers correct output is very, very important.**
- **The requirements can and should be known in detail up front**.
- **<u>The requirements change rapidly and can't be known in detail up front</u>**.
- **Projects can be large and communication paths complex.**
- **<u>We don't have to prove that we did what we promised.</u>**
- **We do need to track costs and results to help demonstrate that what we are doing is cost-effective.**

Favorite Practices

Typically, in-house software development groups use practices not dissimilar to the methods used in contract groups. This may be more of a historical effect than driven by their particular needs.

> **Lighter-weight Documentation:** In-house groups often produce less documentation and have fewer required documents. There's *no need to prove that a contract was fulfilled*, no auditors to satisfy, and the staff doesn't turn over as fast as that of contracted software so it is possible to carry information in people's heads. However, the *larger projects* still require a certain amount of documentation.

CMM: The in-house groups are less likely to go after formal CMM certification since they aren't trying to sell themselves, but appear to be using CMM methods frequently from what I've seen, often to attain *delivery within budget* and to *track costs and results*.

Formal Requirements and Analysis, Unified Modeling Language, Rational Unified Process: Not surprisingly, in-house and contracted companies that are working on similar projects will often use similar methods to ascertain *requirements in detail* and attain *correct output*.

XP and other agile methods: Agile methods have been pioneered on custom in-house projects, and the authors write persuasively about dealing with *rapidly changing or uncertain requirements*. Most agile methods assume a small project size, colocated development staff, and very easy access to the user, all of which are common but not universal attributes of in-house projects. Agile methods are also easier to implement in this practice culture than in the contract culture because the projects don't all have to fit into a fixed-price bidding process, which assumes that a project can be accurately estimated at its beginning.

Drafting Inexperienced People: I've been regularly alarmed by stories at the Software Testing and Reliability conference that tell of assigning people from other, unrelated jobs to head up testing for projects such as insurance software, or of hiring developers who are obviously seriously underprepared to take on the job. While this undoubtedly goes on in all practice cultures to some degree, it seems to be more common when the hiring organization's main business is not software-related.

Commercial Software

Business Model

Commercial software is software written by one company and sold to other companies for use in their own businesses. Retailers or other middlemen may sell the software off-the-shelf, or it may be sold directly to the customer. It may be strictly a piece of software, or may accompany a particular piece of hardware as processing and control software. If extensive customization services are included in the sale, rather than just the software itself, then the company belongs to the "custom software under contract" practice culture, since the contract ends up driving a lot of the economics.

In the simplest terms, the profit on this software is the difference between how much it cost to make, and how much it can be sold for – which is the product of sales volume and price per item. Sales volume depends on the attractiveness of the product in a competitive market. Attractiveness of commercial software is based on features, reliability, and service – glitzy new features for the sake of being new are not generally rewarded, particularly when selling to large companies. In fact, IT departments are notorious for not wanting frequent new releases, which they would have to test and integrate into their existing support structure. Price per item is subject to competition, so it can't be jacked up infinitely. The math works out to suggest that moderate budget overruns can be made up eventually by sales volume, which is certainly what I've observed. (This analysis assumes that there is competition – presumably a monopoly can behave differently.)

There are market windows for commercial products, often based on trade shows, but they are not as frequent or as powerful a force as the consumer market windows. "The need to develop and refine a central set of functions that operate correctly and efficiently drives the releases of system products like Windows NT and Windows 95; the target ship date is of secondary importance." (Cusumano 1995)

Typical commercial software includes database management systems, operating systems for commercial use, network management software, and drivers for peripherals used mainly in a business setting such as high-end laser printers. Medical and avionics software includes medical records systems, diagnostic programs, and air traffic control systems. Despite Microsoft Office's undoubtedly large sales figures, I don't believe Office is an archetype for this practice culture. I say this because I don't see commercial software developers flocking to practices

espoused by Microsoft – in fact, there seems to be a healthy dose of skepticism. One might also note that Office is purchased routinely by home users, whereas enterprise network management software is not.

The *regulated industries* are a subset of the commercial software vendors that work in areas regulated by a federal agency, generally to ensure the safety of the citizenry. Medical products (regulated by the Food and Drug Administration) and aviation software (regulated by the Federal Aviation Administration) are two of the biggest players in this culture. While these two business areas don't generally think of each other as sharing a common culture, I've observed a surprising level of similarity.

The Situation

- **Criticality:** There is a range from life-critical to mission-critical to mission-important. Medical products and airplanes are usually life-critical, operating systems and network software are mission critical, and printer drivers are merely important. Regulated products generally are life-critical.
- **Uncertainty of Users' Wants and Needs:** The requirements for products aimed at an existing business activity are generally reasonably well determined, but not as specific as the requirements for custom software. The users in the regulated industries tend to be very conservative, which results in a low level of uncertainty. For medical products, everything from data accuracy and precision to report formats must match earlier precedents. Aerospace software requirements are dictated by the specific equipment being controlled, so again there are considerable specific requirements. Getting approval for a completely new device in a regulated area is incredibly painful, so the number of experimental new products is, I think, less than in the non-regulated businesses.
- **Range of Environments:** The environments are the sum of all environments used by the companies to whom you wish to sell – this is a larger set than what's used in a single company, but tends not to include everything possible. For instance, compatibility with the America OnLine browser is unlikely to be a requirement in an online payroll deduction package, whereas compatibility with multiple recent versions of Internet Explorer and Netscape is. In the regulated industries, it is common to limit the environments in which a product is warranteed to run – this reduces the reliability problem space to a tractable size.
- **Cost of Fixing Errors:** The cost of fixing errors in released software is significantly higher than in the custom-software practice cultures. The software has moved into many people's hands, so patches must be distributed widely – this has gotten easier with the advent of the Internet, but it's still not trivial. Some products sell a maintenance contract to cover the cost of distributing upgrades, while others give them out for free. In the regulated industries, patches have to be distributed to all affected parties, so records of who owns what have to be kept for years. Worse, if it's a serious bug, the patches have to go out right away. Just because a target system is now sitting in a country with unreliable phone service and light-fingered customs agents does not excuse the vendor from getting the patch to the customer somehow, some way.
- **Project Size:** Projects can be small, or fairly substantial. Aerospace projects seem to run larger than medical products, probably because airplanes are bigger than your average medical device.
- **Regulation:**
    - **Non-regulated industries:** Generally there are no required process audits by any external agencies, including the purchasers. There are industry standards in some businesses such as telecommunications, which are often set by a consortium of the companies themselves. The customers have high expectations for correctness and reliability of the software, and the larger customers have some serious clout both financially and legally if the software doesn't live up to expectations. Extensive and detailed tracking of post-release problem reports seems to be common.
    - **Regulated Industries:** Many people think that practices in the regulated industries are dictated in detail by the federal government. This is in fact not true (unless working under government contract). There are some required practices, such as hazard analyses in the design stage, but the primary expectation is documented compliance to one's own stated practices, whatever they may be. There are extensive audits to demonstrate compliance. Each and every serious post-release problem must be reported to the vendor and the vendor must demonstrate both tracking and resolution of the reports in audits.
- **Communication between Developers:** Most companies appear to operate in what I think of as the "default" mode – the same person designs, writes, and unit tests a piece of code, and most people are permanent

employees rather than contract employees. These practices make both concurrent and forward communication simpler than in the preceding practice cultures.

- **Organizational Culture:** Varies, but control cultures seem to be less prevalent than in the custom software market.

Assumptions

The situation and capabilities again lead to some common assumptions:

- **Delivery on time is important, but budget overruns can probably be made up.**
- **The customers expect correctness and reliability.**
- **The software must work in a variety of different environments.**
- **We are expected to know about and solve problems after release.**
- **(In regulated industries), we need documentation, and we need to do what we say we do to pass audits.**

Favorite Practices

The commercial software vendors seem to be in a middle ground between the purveyors of custom systems and the mass-market software vendors. There are some large vendors whose staff has published extensively (IBM, Kodak, Rockwell), but many of these also operate in the contract software practice culture. It can be difficult to tell if a particular piece of work from these sources is independent of the contract culture. There is little published that identifies itself as clearly from the commercial software practice culture, and practitioners complain that there is little written about them. This may suggest that the total number of practitioners isn't very high, or maybe they just don't write much. It may also be that the commercial vendors don't benefit from revealing their practices to their competitors, whereas the Department of Defense benefits from spreading its favorite practices to more potential contractors.

Key Customers. The definition of *correct behavior* for Hewlett-Packard's cardiology products was heavily influenced by the opinions of a few key customers, who were themselves influential cardiologists. These key customers were frequently consulted during design and participated in field testing of new devices. I have heard of similar practices in other companies as well. These key customers are in a long-term relationship with the company, rather than being randomly selected as is done for focus groups.

"Real" Beta Test. Commercial software typically undergoes field trials when there is still time to fix the problems that are found. HP cardiology software underwent a series of field tests in doctors' offices under very close scrutiny by both developers and medical personnel – often the software was instrumented to keep a complete record of everything that went on, and any errors had to be fixed immediately. This practice was considered part of ensuring *correctness and reliability*.

Release Criteria and Quality Planning. I've observed quality planning used extensively in Hewlett-Packard for commercial software in both the regulated and non-regulated fields. In the regulated fields (now part of Agilent or Siemens), release criteria had to be set well before the release date, and the product had to pass its own release criteria. As a result, the release criteria were carefully chosen to be both achievable and sufficient to ensure *correctness, reliability, and safety*. In both fields, formal quality planning is used to decide what techniques and practices will be employed to achieve the intended levels of quality – everything from requirements to design methods to testing practices are carefully considered for efficacy and efficiency. Practitioners at other medical companies have told me that they use similar practices, and I believe this to be a fairly widespread practice in commercial software.

Inspections and reviews. Inspections and reviews appear to be heavily used in both regulated and non-regulated fields, because of their demonstrated effectiveness in achieving *correctness and reliability*.

Total Quality Management. TQM is still a favorite in the conservative regulated industries, because of the emphasis on managing practices by measuring their results, and the traceable documentation structure, which *supports audits*.

Tailored versions of CMM and ISO 9001 practices. ISO 9001 certification is expected in some areas of this industry – while this may not actually improve software practices, it does help with setting up the necessary record keeping. In the regulated industries in particular, *defects in the field are religiously reported*, which provides a substantial amount of information on which to base the choice of practices from the CMM or other models. A pragmatic approach is quite typical, so CMM-style management practices may be used to achieve *delivery on time*, but certification isn't sought unless there is some reason to think that the software buyers are interested in it. It is not uncommon to voluntarily seek out certifications such as ISO 9000 if this appears likely to confer an advantage over competitors or if the certification is required in some country in which the software will be sold.

Compatibility, Configuration, and Usability Testing. Due to the *variety of both users and user environments* supported, the commercial software vendors often spend more time and effort on these types of testing than the authors of custom-written software.

Variety of lifecycle models. The contractual issues which demand requirements up front are not present in commercial software so there's room for some variety. Methods that are very light on documentation such as XP are not popular in the regulated industries because the methods don't produce the *documentation to support audits*. I've seen staged delivery used with considerable success, and forms of evolutionary delivery as well. Teams working with hardware groups often have a high-level plan in a simple waterfall format in order to synchronize their work with the hardware development.

Mass-Market Software

Business Model

Mass-market software is software for home or small business use. (Firmware is in a category of its own because some of the situational factors have a huge impact on the business model, as we will see later.) The mass-market world is very different compared to any of the preceding cultures. Moving from a regulated or contract-driven organization into a mass-market organization can produce prolonged culture shock. At first glance, the practices used in this industry can appear crazy – but on further investigation, there are assumptions behind the behavior that do make sense.

The software is usually sold via a retailer or other middleman to the consumer, rather than purchased directly from the company that wrote it. The mass-market software market appears to be driven by market behavior which predated the invention of consumer software by quite a few years – frequent releases of "new and improved" items, heavy advertising to convince consumers to buy software which is discretionary at best, and a reliance on seasonal market patterns. If you miss Christmas or start of school, you can lose a huge percentage of sales for the entire year. Releasing new, cool features before the competition is a very strong factor in this market.

Typical mass-market software includes tax-return software, greeting card programs, games, and drivers and associated software for mass-market peripherals such as low-end inkjet printers and digital cameras. I hesitate to put mass-market operating systems in this category, since Microsoft is by far the largest player in the field and they seem to have at least one foot in the commercial software practice culture (as described by Cusumano 1995).

The Situation

- **Criticality:** If consumer software fails, people lose time and are often annoyed, but seldom lose large sums of money and are rarely at risk of injury or death. (Note that this refers to *software*. Failures in the firmware in your car may well be able to kill you.)

- **Uncertainty of Users' Wants and Needs:** The customers typically are a mixed bunch and difficult to contact. It can be quite challenging to get a truly representative cross-section. The most extreme case is a brand new type of product, where the customers can't tell you what they want because they've never even thought of doing that with software before. For products that have been around a while, the end users may be pretty sure what they want, but the marketing department usually makes up for it with lots of dandy new ideas.
- **Range of Environments:** Mass-market software has by far the widest range of environments in which it must operate, and those change constantly, making high levels of reliability very difficult and expensive to achieve.
- **Cost of Fixing Errors:** The software support life is quite short compared to commercial software. Patches are often distributed via the Internet now, which has reduced the cost of patch distribution quite substantially. (However, there is a practical limit on how large an upgrade can be downloaded via the Internet, with which Microsoft is continually flirting.) The vendor is under no obligation to find and notify the affected customers.
- **Project Size:** Projects generally run between one and a few teams, and are usually less than a year long. Larger projects require extremely high sales volumes to recoup costs at consumer-level prices.
- **Regulation:** No audits, no comprehensive regulation – in fact, unlike the other businesses, it is common to include a warranty disclaimer in the End User License Agreement, which must be accepted in order to install the software. While the legality of such a disclaimer is doubtful (Kaner 1998), the situation basically boils down to little or no regulation.
- **Communication between Developers:** As in commercial software, developers typically do all the work on "their" code. Geographic separation of developers seems to be more common in this environment than in the others, although my evidence for this is purely anecdotal.
- **Organizational Culture:** Varies, but the control model doesn't seem to be prevalent. Competence cultures (such as Microsoft) are more common, and cultivation cultures appear in the smaller companies.

The situational factors lead to a set of assumptions that are very different from the other practice cultures:

- **If we don't hit the market window, we're dead.**
- **This is a competitive market.**
- **The requirements are ill-defined and vary widely.**
- **If we satisfy 80% of our customers, that's good enough.**
- **If we ship with bugs, we can fix them later (as long as they're not too bad).**

Favorite Practices

The traditional, documentation-heavy methods designed for enormous multi-year projects simply don't work in this practice culture. Sometimes this has resulted in organizations ditching *everything* from the older cultures, and starting anew with a code-and-fix approach. This, coupled with the prevalence of just-out-of-school programmers in this culture, sometimes results in entire organizations that cannot choose to adopt potentially useful practices because literally no one in the building knows about them. Sometimes the practitioners rediscover or reinvent a practice on their own, and you'll see a breathless announcement of an invention that bears a very strong resemblance to practices in use for years in other practice cultures. To be fair, adapting a practice to a radically different situation is harder than simply adopting one by the book, so it's not surprising that the transfer of practices into the mass-market practice culture from other cultures is often not high.

Some popular practices are:

Focus groups and formal usability testing. These techniques to ascertain the *ill-defined and varying requirements* are heavily used by the larger consumer software vendors.

Lightweight planning and project management. Projects often must complete within a few months in order to *hit a market window* or *beat the competition*, so they tend to be shorter and smaller, which requires less formal planning.

Lightweight documentation. The requirements are usually not written out in detail, and design documentation may be almost non-existent.

"Good Enough" decision-making. The "Good Enough" movement is based on the idea that defect-free software in the mass-market situation is both unattainable and likely unnecessary, and focuses attention on planning to "understand the problems and benefits of a situation well enough to eliminate (or prevent) the *right* problems and also deliver the *right* benefits." (Bach 1997). The definition of "right" is based on both customer needs and business realities – *satisfying 80% of the customers* is often good enough. This is a difficult practice for many practitioners in this and in other practice cultures to accept, since admitting that software does ship with defects is often equated with "not valuing quality" or otherwise not being on board with official company goals, so the adoption of "Good Enough" methods has been slow. Oddly, the commercial software vendors use release criteria and quality planning to make decisions in ways that seem very similar to the "Good Enough" decision-making practices, although the intended reliability and correctness levels are usually considerably different.

Rapid Testing, Exploratory Testing, et. al. Many consumer software vendors use these testing techniques, which look for the important bugs first, rather than systematically testing against an extensive list of requirements (which doesn't usually exist anyhow). Since *the requirements are often not well known*, exploratory testing techniques are used to look for user dissatisfiers.

Agile Practices. These are designed specifically to work with *ill-defined user needs* that tend to change during the development period. Agile methods are actually easier to use on in-house projects, such as the C3 project described by Beck (Beck 2000), than on mass-market projects, because it's much easier to talk to the users frequently. However, agile methods are also used in the consumer market, as described in Iansiti and MacCormack's work (Iansiti 1997). There's also a strong tendency in the mass-market to claim that eXtreme Programming is being used when the only XP practice in use appears to be the lightweight approach to documentation.

Code, then Test. This is the practice of finishing all the coding first and then fixing all the defects – this doesn't preclude up-front design so it isn't quite the same as McConnell's code-and-fix. (McConnell 1998). The practice of coding everything before fixing anything tends to be a really bad match for the mass-market culture since the work piles up at the end and makes the release date uncertain, which this culture can ill afford due to the *fixed market windows*. The only reason I can think of for the persistence of code-then-test is that it gives the illusion of rapid progress towards the all-important release date, at least during the coding phase.

Firmware

Business model

The software is made as an adjunct to a physical product, such as a toy or automobile or printer. The products themselves are subject to market forces that depend on whether they are mass-market products, commercial products, and so forth. However, the situational factors are dramatically different from those of the other practice cultures, and I think they overwhelm the effects of the market forces. For instance, I found that the practices for software used in HP's cardiology division and its inkjet printer division were substantially different, but the firmware practices were quite similar.

The Situation

- **Criticality:** The products run the gamut from toys to automobiles to missiles, so some are life-critical while others are not.
- **Uncertainty of Users' Wants and Needs**. The requirements of firmware are set by the hardware for which it is developed. Generally the requirements are easy to discover by asking the hardware developers, who are readily available. However, the requirements do change as the hardware design develops over

time, and there is a common practice of expecting firmware workarounds for hardware problems discovered late in the game.

- **Range of Environments**: Firmware is generally expected to work in only one environment – the hardware that is using it. However, if that hardware connects to hardware or software that varies widely (like a specific printer connects to a wide variety of PCs), the firmware will probably have to handle some of the variation itself**.**
- **Cost of Fixing Errors:** Fixing a firmware error requires re-flashing the firmware or replacing the entire chip, which in turn requires manufacturer or service personnel getting their hands on the product. For anything sold in the mass-market, recalling millions of items makes this prohibitively expensive. For items with lower sales volume, simply finding the items may be practical, but all the shipping and work on them is economically practical only if the item costs thousands of dollars. This is a huge factor in driving software engineering practices for firmware developers – you simply cannot afford mistakes, even on low-criticality products.
- **Regulation:** Firmware in medical and aviation products is regulated just like the commercial software in those fields. The regulatory picture for mass-market firmware is considerably different than for mass-market software – you are not expected to accept an "End User License Agreement" excusing an auto manufacturer from responsibility for the results of a firmware defect in your car.
- **Project Size:** Firmware has historically been smaller than software because it has to fit in permanent memory, but in the last decade the chips have gotten a lot bigger and the size of firmware projects has increased. The amount of code in an automobile, which is spread over multiple processors, is both amazing and a little scary.
- **Communication between Developers:** Most firmware developers seem to be permanent employees and I have not heard of any instances of the junior-senior division of labor.
- **Organizational Culture:** I have no real information on this.

The situational factors lead to a reasonable set of assumptions:

- **The firmware must be of high reliability and correctness.**
- **We have to work closely with the hardware developers.**
- **The hardware will change during development.**

Favorite Practices

Waterfall Lifecycle. Firmware developers often use a waterfall lifecycle because they *work closely with hardware developers*, whose work is usually sequential rather than iterative, and a waterfall lifecycle is easily synchronized with that.

Realtime Design Methods. Firmware developers use a number of design methods that are not needed in other fields to deal with realtime input from the devices that the firmware controls.

Evolutionary Delivery. Surprisingly, some firmware teams do use some varieties of agile methods, wherein the hardware team plays the role of the customer. This seems to help with the *changing hardware requirements.*

Inspections and reviews. Firmware developers, even for mass-market products, tend to make heavy use of inspections and reviews to achieve *high reliability and correctness*.

Testing by developers. Firmware teams seem more likely to be heavily involved in testing than their software counterparts. In some cases, this is driven by safety issues – a partially assembled or prototype piece of hardware may not be safe for an untrained person to handle. There is also a big emphasis on "white-box" testing to help achieve a high level of *reliability and correctness*, in which a development background of some sort is a necessity.

Open Source Software

A lot of the situational variables are clearly quite different for open source. I'm not personally familiar enough with either the field or the writings about it to do an analysis on open source.

Academic Software

Academic software is written by people in universities to do research – this would include both computer science research and research in many other fields such as biology, chemistry, psychology, etc. Programs used to do research in fields other than computer science are often in use for many years and there can be considerable turnover of the graduate student authors, who may be entirely self-taught. I was first introduced to programming as a chemistry graduate student in the 1980s, when I worked on an extensive set of FORTRAN programs for various types of computational chemistry. At the time, the graduate student authors had very little training and no experience in aspects of software engineering such as structured design or testing. "Favorite practices" were very primitive and sometimes counter-productive (such as the prohibition against indentation favored by one professor). Things have probably changed since then, although it's doubtful that graduate students in other fields have picked up a complete computer science education.

Internet Software

Business Model

The Internet is a technology, not a business model, and as such the technology appears in multiple business models – some closely resembling business models we saw above, and some that are new.

- **Custom Contract Software.** Plenty of people are writing Internet software for other companies – the main difference between this and traditional custom contract software is that the customer companies and the contractors are often much less experienced at drawing up contracts than the traditional customers and contractors in this business model, so the contracts are much less detailed. The software written under contract can be any of the types described in the following business models.
- **In-House Custom Software**: This is generally business software for use within the company - payroll, health insurance, etc. The economic factors for Internet software intended for in-house use seem to be pretty much the same as for more traditional client-server software – the benefits have to justify the cost, and projects can be cancelled or delayed due to budgetary constraints. The IT groups do face some issues with learning so much new technology, which drives some of them to hire or contract with outside programmers.
- **Commercial Software**: Internet software written by one business to be used by another business mostly is sold as "services" to which the customer company subscribes. Some are invisible components of custom services, and others are quite visible to the users, such as services that manage health benefits information or charitable payroll deductions for multiple companies.
- **Mass-Market Software:** There's been a lot of software written for consumers but very little success at extracting money directly from consumers. This is not the same business model as we find in traditional mass-market, where the consumer has to pay for the software – I am not sure what this will become.
- **E-Commerce**: E-commerce sites used by mass-market consumers represent a new business model, where the users are mass-market consumers but the people who pay for the service are not. This strongly resembles the traditional mass-market business model, where time-to-market and competitive new features are more important than cost. Most e-commerce sites allow the consumer to buy a product or service (books, airline tickets, clothes, computers) and a percentage of that transaction pays for the service. Others operate as inducements to buy a product, rather like a type of advertising.

The Situation

- **Criticality:** In-house Internet applications that replace existing applications are expected to meet the same standards for correctness, although the tolerance for availability and usability may be lower than usual.

Consumers expect e-commerce applications to keep track of all the money and orders without fail, so expectations of correctness are quite high, although the customers will tolerate performance that would be unacceptable in shrinkwrap software. For the first few years, the consumer-facing software was being used mostly by "early adopters" who were willing to tolerate a substantial number of problems, but as the technology becomes more widespread, expectations of reliability from consumers are increasing dramatically. The availability requirements can be more stringent than in traditional client-server applications, because it's very easy to pick up customers in multiple time zones, and suddenly you have no "middle of the night" maintenance windows.

- **Uncertainty of Users' Wants and Needs:** For both commercial and consumer-facing software, the service being provided is often either completely new or was previously done in a very different manner, so the uncertainty of requirements is often very high.
- **Range of Environments**: The range of environments depends on whether the end-users are people working at a company with its pre-determined environment and tasks to be done, or are consumers with a wide range of environments and tasks. The number of possible combinations is actually larger than that of shrinkwrap software, because the software must be compatible with not only the operating system but also the browser.
- **Cost of Fixing Errors:** The cost of releasing new functionality and bug fixes to a website is in fact quite low compared to other forms of releasing software fixes. However, changes to improve performance or support loads that increase by orders of magnitude have not proven to be as simple – they often require tearing apart part of the infrastructure and redoing it.
- **Regulation:** The in-house projects are subject to the same regulatory and audit requirements as those of other in-house applications. The consumer-facing software is not actually purchased by the consumer, so there is no End User License Agreement in force. It is unclear at this point what level of responsibility will play out in the consumer market.
- **Project Size:** Project sizes range from quite small to considerable for the larger commercial sites.
- **Communication between Developers:** I am frequently seeing small development groups scattered across wide geographies, but I don't have enough data to say if this is a trend or not.

The assumptions based on the situation and business model appear to be heavily affected by a third factor: the home practice culture of the people in question. For all practice cultures, some of their accustomed assumptions work and some don't. I have seen some individuals and teams consciously inventory their own assumptions and figure out which ones should be kept and which replaced, and others throw the baby out with the bathwater. The latter is more common with teams consisting entirely of bright young people with a command of the technologies but little project experience, although more experienced people can also fall prey to the illusion that the "Internet is nothing like anything that has ever been done before".

Immigrants from the in-house custom software practice culture often need to:

- Discard the emphasis on low cost over time-to-market. This emphasis makes sense when supporting internal efficiency measures, but it holds back the rapid progress needed to succeed in this market. Often teams are not consciously aware of which of their company policies or standards are designed to keep costs low.
- Keep their skills with handling large disparate systems: configuration management for systems, change management, database design, security, proactive maintenance, etc. Immigrants from practice cultures that didn't make large systems often don't have these specialized skills.
- If not already using agile methods, investigate switching. Collection of requirements in detail up-front does not seem to work in this business model.

Immigrants from the mass-market practice culture often need to:

- Learn the skills associated with large disparate systems.
- If not already using an appropriate project planning methodology, learn one of the agile methods.
- Plan to learn methods for producing high-reliability code – you'll need it soon.

All the Internet immigrants, regardless of their original culture, seem to have a tendency to drop practices to which they are accustomed because of the compressed schedules typical in this field. The results are not as bad as one

might think, because to date the focus on time-to-market and low customer expectations for reliability have made taking some shortcuts reasonable. However, this is probably not going to last as expectations for reliability increase.

Internet Immigrants Learn From Each Other

It is not uncommon for immigrants from more than one practice culture to work together on the same project - many companies contract out applications or hire complete teams of new programmers to work with existing in-house staff. The meeting of minds can be antagonistic or mutually beneficial, depending on the attitudes of the people involved.

A good example of a mutually beneficial relationship is Bill Tuccio's story of a joint project. (Tuccio 2002) Bill worked for a dot.com company that contracted with a theater chain to create a system to sell movie tickets online. The dot.com folks worked directly with the theater chain's IT department. Initially there were some difficulties. The IT people at first didn't take extra steps to push the infrastructure orders through – they didn't see the artifacts that they were accustomed to seeing, and assumed the project couldn't possibly meet its "aggressive" schedule. Eventually the two groups started having twice-weekly status meetings and frequent phone calls. It became clear that the dot.com staff was progressing very quickly through the software despite their lack of written documentation, and the IT staff responded by speeding up their part of the work. The dot.com group also found some advantages in the IT folks' methods. At one point, the IT people sent an expert to do a risk analysis – something that no one in the dot.com really knew how to do. The risk analysis identified the need for some extra staffing that turned out to be a very good idea. Overall, the relationship wasn't always smooth, but in general the staff of the two companies took the time to ask "why are you doing that" and adjust their assumptions and their practices as needed. By the end of the project (which did release on time), both sides had learned some useful things from each other.

Evaluating Practices for Effectiveness

Now that you've visited a number of practice cultures, let's return to the original topic of this paper – how can I evaluate a practice that I'd like to adopt? Once you've understood the basics of the practice, the next step should be to determine the assumptions behind the practice, in order to compare those assumptions with your environment. To find the assumptions, you need to know where a technique originated, and in what other contexts it has been used.

Sometimes the description of a technique will also describe the business in which the technique originated – this is more common now than it was in the 1970s, when there were fewer practice cultures. Sometimes the examples and context surrounding descriptions of the technique will identify its origins. Other times, you may have to look for the professional history of the author, which nowadays is often available on their website. Articles in magazines and journals are usually aimed at the practice cultures served by that publication. Sometimes the publication is quite upfront about its audience. Crosstalk identifies its audience right in its title*: Crosstalk: The Journal of Defense Software Engineering*. If the magazine isn't quite that forthcoming, look for their message to potential advertisers. Software Development Magazine: "Discover the many ways we can deliver your message to over 100,000 corporate software development leaders every month". (SDM 2002)

Once you've identified the practice culture in which a practice originated, you can examine the benefits attributed to it against the business model and situational factors for that practice culture. Ask yourself: are any of the benefits dependent upon assumptions or factors that won't be true in your situation?

For example, Sue's company is working on e-commerce products. They are interested in increasing the reliability of their software. Sue knows that medical products are highly reliable, and she also knows that detailed documentation of test plans and procedures is typical in the medical products field. Sue wonders if introducing IEEE 829 standards for test documentation would be helpful.

What are the assumptions behind the use of detailed documentation in the medical field? The medical products practitioners will readily tell Sue that written test plans and test procedures lead to thorough testing, which results in high reliability. Many practitioners will tell Sue that everyone, in every field, should document their test plans and procedures in detail. However, the practitioners may neglect to tell Sue that some of the documents are primarily intended to satisfy FDA audits. People who are not directly involved in the audits may not even realize this. Sue

also may not discover that not all medical products companies follow the IEEE 829 standard exactly as written. Most importantly, Sue may not discover that using the IEEE 829 standard did not teach the medical products testers which tests are useful and which tests are not. They learned that someplace else, which may not be as visible as their use of IEEE 829 standards.

Introducing IEEE 829 standards to Sue's group may improve reliability if they haven't had a systematic method to communicate what to test, what results are expected, and what has already been tested. However, following the entire IEEE 829 requires a great deal of documentation and will probably be overkill in Sue's situation. In fact, if Sue's project has a fixed ship date, precisely following the IEEE 829 standard may actually reduce reliability by spending time preparing documentation that could have been spent on finding bugs.

Once you start looking for the assumptions behind the use of a practice, you'll ask much better questions about the context or situation in which the practice is used, which will clarify the ways in which the practice delivers value in that situation. This insight can help you decide whether to adopt a practice as-is, modify the practice, or stay away completely.

Summary

The practice cultures described in this article are by no means definitive. Despite the length of this paper, I feel I've only scratched the surface of this topic. Undoubtedly each of you will have your own observations about business models, situational factors, and practices, based on your own experience. I've tried to demonstrate the connections between assumptions and practices in order to give you some idea of why some practices seem to "fit" your own situation while others seem ineffective, if not downright stupid.

If you're interested in talking to people from other practice cultures, you'll find them at conferences, online, in classes, in the pages of magazines and journals, perhaps at lunch if you work in a large company with a varied business. You'll learn many more interesting and useful things from these people when you understand that their assumptions don't necessarily match your assumptions, and there may be good reasons for that. Listen, ask questions, and don't jump to conclusions too fast. Remember, it all depends.

References

Bach[1997]: Bach, James; "Good Enough Quality: Beyond the Buzzword"; IEEE Computer; August 1997. (also available at www.satisfice.com)

Bach[1999]: Bach, James; "Good Practice Hunting"; Cutter IT Journal; 1999. (also available at www.satisfice.com)

Beck[1999]: Beck, Kent; *eXtreme Programming Explained: Embrace Change;* Addison-Wesley; 1999.

Cusumano[1995]: Cusumano, Michael; Shelby, Richard W.; *Microsoft Secrets*; Simon and Schuster; 1995.

Highsmith[2002]: Highsmith, Jim; *Agile Software Development Ecosystems*; Pearson Education; 2002.

Humphrey[1989]: Humphrey, Watts; *Managing the Software Process*; Addison-Wesley; 1989.

Iansiti[1997]: Iansiti, Marco; MacCormack, Alan*; Developing Products on Internet Time*; Harvard Business Review; 1997

Kaner[1998]: Kaner, Cem; Pels, David; *Bad Software: What to do When Software Fails*; John Wiley & Sons; 1998.

McConnell[1996]: McConnell, Steve; *Rapid Software Development*; Microsoft Press; 1996.

Schneider[1994]: Schneider, William. *The Re-engineering Alternative: A Plan for Making Your Current Culture Work*. Irwin Professional Publishing, 1994.

SDM[2002: http://www.sdmagazine.com/sdonline/mediakit/, Aug 16, 2002.

SEI[2002]: http://www.sei.cmu.edu/about/about.html, July 1, 2002.

SET[2000]: http://source.asset.com/stars/loral/cleanroom/tutorial/, dated 2000, Software Engineering Technology, Inc.;

Tuccio[2002]: Tuccio, Bill; personal correspondence.