

# Handling the Endgame of a Project

Warren Craycroft, ProjectConnections staff

## ***What is the endgame?***

There is no hard, fast definition of a development project's endgame. In general, the "endgame" is the set of activities that occur late in a development project as the design approaches the point of being released to revenue customers. Design methodologies such as spiral development, incremental delivery, evolutionary delivery, etc, may blur the distinction of a product's release date as the "end" of development. But, in general, the endgame has the following characteristics in any development project:

- It is late in the design stage. Major changes in the design are not anticipated. Some prototypes may exist in some state of "working". There may be an alpha or beta program in progress.
- A release date has been officially announced, or there are well-formed expectations within the company on when release will occur.
- Other groups in the organization such as manufacturing, service, marketing, and finance are making commitments of time and resources to prepare for release.

## ***What are the typical issues you encounter?***

### **Breakdown of process**

As a project enters the endgame period, it comes under increasing pressure to abandon process and "break for the finish line". This pressure to cut process short is not limited to organizations with a weak development process. Even industries involved in heavily regulated development of safety-critical systems are subject to the same endgame stresses. Most project managers have experienced the following endgame process breakdowns:

- Quality targets are compromised. Test plans are altered; Pass/Fail testing criteria is modified.
- Reviews and audits are skipped, or become perfunctory and of little value in influencing and moderating development.
- Critical requirements such as key product features and COGS targets are no longer carefully monitored by project management.
- Project status information from project management to executive management becomes imprecise and unreliable, or the development organization goes "opaque" and stops delivering meaningful status altogether.

## **Schedule becomes difficult to manage**

In the endgame period, different company organizations are synchronizing on the development schedule and preparing their own functional areas on timelines that mesh with development milestones. But just when accurate development schedule information becomes critical for this business synchronization, the development schedule begins to deteriorate. New tasks appear out of nowhere, milestones are missed, and dates become moving targets. The development schedule may enter the "runner's nightmare": no matter how close you get to the release date, it is always 3 weeks away, continually receding as you run faster and faster toward it. The schedule may even suffer a complete meltdown. A release date thought to be 3 weeks away is discovered literally overnight to have a 6-month slip! Such schedule meltdowns are altogether too frequent; a few have been very public.

Project managers who find themselves in schedule deterioration often counter with increased scheduling tool usage. They redouble their efforts to track their projects in greater detail, increasing task resolution down to the day and hour level and modeling task dependencies at a corresponding level of detail in their Gantt charts. But rather than regain control of the schedule, such scheduling tool efforts often exacerbate the situation instead.

## **Minutia floods**

As the endgame progress toward release, the number of issues generated by the project tends to increase exponentially. These issues include bugs in the design, flaws in the requirements, test results that cannot be interpreted as passes or fails, transient bad design behavior that cannot be recreated or whose solution is not yet clear, late enhancement requests, etc.

As the flood of issues increases, critical knowledge about the current project state becomes harder to determine by project management. In turn, it becomes difficult for the development team in the trenches to determine priorities among the possible tasks on which to work. And so they may work on the tasks in an arbitrary order, or do the ones that are championed by the loudest voice. It can even become difficult to figure out what "done" means, and how close they are to getting there.

As problems arise, they may be recognized as problems that have been seen before, and were even previously resolved. But no one can remember the details of the solution or the decision process. And so the problem is re-solved, the decisions re-decided, the solutions re-discovered. Sometimes, these problems become intrusive old friends, dropping in occasionally, unannounced, to create a little more havoc in an already painful endgame situation.

## **Team behavioral issues**

Team behavioral issues are often a cause for endgame problems and even outright project failure. At least one study suggests that in recent years team behavioral issues have become a predominate cause of project failures. These issues include low morale, poor individual productivity, lack of employee commitment, and poor interpersonal relationships.

## **Launch desynchronization**

Inaccurate schedule information can cause the company-wide launch schedule to become desynchronized. Money may be wasted on the premature acquisition of resources. For example,

Manufacturing may commit to floor space, staff, and capital equipment based on a release date that cannot be met. These resources may end up sitting idle at significant cost. Marketing may make a premature new product announcement to its sales force or its customers, which kills the current product line. If such an announcement is mistimed, it can have devastating effects on the sales force morale (and headcount) as well as on customer credibility. Investor Relations may brief stock analysts with product delivery expectations that cannot be met, causing an investor backlash.

## **Premature product release**

The endgame issues described above can escalate to the point where the pressure to release the product becomes overwhelming and sweeps aside concerns about quality, fitness for use, and in some cases even product safety. The prematurely released product may generate some immediate relief for the company. But, in the long run, the premature product may have quality problems that have to be fixed under the noses of customers. This can generate returns, lost sales, and even damage to the company name and reputation.

It is tempting to conclude that such premature release decisions are "bad" decisions, made by shortsighted managers who ignored the concerns about quality and safety and chose to "make their numbers" at the expense of long term product issues. But in all too many of these endgame situations, executive management is given no hard data from the development team with which to judge the current state of the product. Left with this void of data, management -- good and bad -- will try to fill in the blanks in the business equation as best they can and make the decisions they have to make. With their development organization having gone "opaque", and their marketing, manufacturing, and finance organization providing forecasts, opportunity data, burn rates, development cost to date, etc, it is not surprising that release decisions may become biased toward release.

## **Premature project termination**

Ideally, projects should terminate with activities that ensure that all obligations of the development team have been met. These obligations typically include the following:

- Customer requirements and obligations have been met.
- All documentation, training, and tools that are needed to manufacture, support, and market the product have been delivered to the groups responsible for these life-cycle activities.
- Any quality plan, ISO9000, CE mark, and any other regulatory requirements for product and project documentation have been met. If audits are part of these requirements, then those audits have been completed.
- The project team members have captured the lessons learned during the project in a form that future project managers can use to avoid repeating mistakes.

Some of these endgame activities are often cut short; the last one is often eliminated completely. A principal cause of this premature project termination is competition for project team resources from other projects, especially those that are newly formed and are building resources rapidly. The situation is exacerbated when the terminating project has slipped in schedule and collided

with the schedules of the emerging projects (the later being in early development phases where maintaining schedules is easier).

Another cause of premature project termination is not recognizing the value of the termination activities. This can occur even when there is no pressure to disband the project. As noted by Kerzner and others, it is the capturing of lessons learned that is often neglected for this reason. Until the value of project debriefs has been demonstrated in an organization, these debriefs, if they occur at all, are likely to be ineffective in truly changing the future behavior of project managers (the “good meeting but then what” syndrome).

## ***What are some solutions to these issues?***

### **Endgame process**

#### ***Laying a good process foundation early***

The best insurance against endgame process breakdown is to write a project plan. The project plan is a roadmap of all planned activities for the particular project. The content of the plan will vary depending upon the industry, the target product, etc. But all project planning should have the following themes.

- All major activities of all groups involved in the project should be at least outlined in the plan.
- Endgame activities should be described in detail and justified with a rationale. The more justification that you can show for endgame process steps and goals, the less vulnerable they will be to endgame pressures.
- The development process should be "gated" with enough checkpoints to keep the project from becoming overextended.

Development organizations that have a written development process should still create a project plan for each individual project. The plan should explicitly call out which activities described in the development process are going to be used for this particular project. The project plan becomes an instantiation of the general development process, tailored to meet the needs of the particular project.

#### ***Plan a "gated" development process***

A principal cause of endgame pressures is the project's becoming "overextended". This means that a project progresses into a subsequent development phase without laying an adequate foundation in previous phases. As the project overextends itself, activities are started too early and commitments are made prematurely. For example, manufacturing may commit to expensive tooling before design risk has been lowered sufficiently. In the worst case, the entire company may commit itself to preparation for a product release date that cannot possibly be met without seriously compromising the product's quality goals.

There is a myriad of development process models. They are described with colorful abstractions such as "waterfalls", "spirals", "incremental deliveries", and "evolutionary development". Some are fundamentally different than others in structure. But virtually all development processes

used in a business environment must have some way of keeping the business synchronized with the development in progress. One way to accomplish this is to have checkpoints, or "gates", in the development process itself. The specifics of these checkpoints will vary greatly with the actual development process, but there should be the following common themes:

- Measure the current technical risk of the emerging design
- Measure the current fitness for use of the emerging design
- Measure current critical parameters such as budget and schedule
- Ensure that the deliverables created and the activities performed to date have laid an adequate foundation for the activities to be performed in the next phase of the project

These checkpoints are implemented in the development process as reviews and signoffs of project deliverables. The project team pauses, examines with management the progress to date against planned goals, and makes a consensus decision on whether the project should continue.

Gating the development process with periodic, intensive reviews allows possible schedule slips, budget overruns, and quality issues to be identified early. Delivering this bad news early gives management some options in realigning the rest of the company with the changing development schedule and helps avoid a painful project endgame.

### ***Lay down a strong vision and requirements foundation early***

This point cannot be emphasized strongly enough. Almost every study of why development projects fail points to poor vision and requirements management as a major culprit.

Capturing the product vision and specifying requirements is the centerpiece of the "front end" of nearly every development process methodology. We offer methods elsewhere to help with this critical planning task. Getting this vision down on paper, obtaining company-wide consensus agreement, and synchronizing the development team on its requirements is a major step toward a successful project.

The deliverable generated by this process becomes a key reference throughout the development process. The endgame development process must keep the critical requirements of a project visible and clearly understood. Although the vision and the requirements can rarely be completely frozen, it should be difficult to make late changes without wide consensus. A number of endgame issues can be avoided with a good requirements foundation AND the ability to manage the inevitable changes as the project proceeds.

### ***Dynamically manage requirements during the entire development process***

As the pace of technology development continues to accelerate, the life cycle of products incorporating this technology continues to decrease. Average development times for high-tech products have been decreasing steadily over the years and are estimated to be about one year. The accelerating pace of technology improvements is also shortening the useful life of products incorporating this technology. As the development time become a large percentage of the product's total lifecycle time, the pressure to improve on a product's design becomes a constant and relentless force impacting the entire lifecycle. This pressure to change the design using improved technology begins during the early design process and continues through the product's development, release, and manufacture. In highly competitive markets, the pace is so furious

that a product is often rendered obsolete shortly after its release, and in some cases even before it has been released.

This constant onslaught of change requests has made dynamic requirements management a vital part of high-tech development. Gone are the days when development could "freeze" a product's design, shut the door on any further marketing change requests, and complete the development of the product in the relative calm of frozen requirements and design. Late change requests from Marketing may be vital for the product's success. The purpose of dynamic requirements management is to keep the door open on change and provide data on the tradeoff of benefit versus cost in time-to-market, COGS, increased risk to the design, etc. for each late change request.

The area of requirement management is a vast subject. Development methodologies vary from specifying 2-page vision statements to the creation of 200-page 'aerospace' requirements documents with consistent use of verb tenses, tagging, etc. However, there are some attributes that should be present in a requirements management strategy to help manage change:

1. A "critical requirement" is an attribute or quality of the product that is vital for the product's fitness for use. If a critical requirement cannot be satisfied by the design, the product should probably not be developed.
2. The critical product requirements should be isolated in a way that makes them easy to track during the project. They can be separated out in a vision statement, or made to stand out in a requirements document.
3. Requirements that mitigate hazards should be included in a requirements document. Note that "hazards" are not limited to human safety or other regulated safety-significant development. A bank transaction software program may have the potential hazards of 1) losing the customer's money or 2) losing the bank's money. Software requirements that mitigate these hazards should be explicitly stated.
4. Any vision statements or requirements statements should be stated in measurable terms. It should be possible (and part of a good validation strategy) to develop a test or review item to verify that the requirement has been satisfied by the design.
5. Each stated requirement should be assigned a status, such as OPEN, FROZEN, VALIDATED, and DELETED. OPEN means the requirement is in progress and subject to change. FROZEN means a consensus agreement has been made to meet the requirement, and it should now hold still. VALIDATED means the design has been tested and shown to meet the requirement. DELETED means the requirement has been removed from the design. This tracking of requirements status individually allows a much more dynamic and flexible treatment of requirements. It more closely models the reality of requirements management, where some requirements are no-brainers and others are difficult to decide upon, state, implement, or test.
6. During the endgame, it is expected that most or all requirements will either be FROZEN or DELETED. Any requested change to a FROZEN requirement addition of a new requirement should be answered with an estimated cost in development time (i.e., impact on time to market), COGS, and general risk to the design (e.g., an extensive software change late in the design).

The depth and detail of requirements documents cannot be generalized. It depends on the industry, the product being developed, the nature of the development group itself, possible regulatory or quality plan requirements. However, the obvious tradeoff should be stated:

- The more you know about the target you are trying to hit, the more likely you are to hit it.
- Increasing the detail of requirements specifications has diminishing returns at some point.

### ***Justify quality goals***

Make clear in measurable terms what the rationale is for quality goals such as test results. Show what the consequences may be in increased risk and decreased quality if a quality goal is compromised. For example, a certain number of pilot production prototypes may be subject to stress testing because statistical models predict production yields based on the number of prototypes tested. During the heat of the endgame, it might become attractive to reduce the number of prototypes tested in order to gain a scheduling advantage. If the number of prototypes to be tested is clearly justified in the test plan with a rationale describing the risk of decreased production yields, then a decision to reduce the number of prototypes will not be made without understanding the risk involved.

Laying out quality goals in the relative calm of early project will help you see your way more clearly to an acceptable level of fitness for use during the endgame, when quality will be assaulted by delivery time, COGS, development costs, etc.

## **Endgame Scheduling**

### ***The true project schedule***

When discussing the difficult job of creating and managing a schedule, we should step back for a moment and discuss just what a project's schedule really is. The project's true schedule is NOT contained on a Gantt chart or PERT chart. Work breakdown lists never list all the tasks that will occur in a project. The true schedule of a project is on the desks, in the files, and in the heads of the people executing the project's work. Gantts and PERTs, work breakdown structures, and tasks lists are useful abstractions to capture, organize, and present the information to a certain level of detail. But if project managers want to determine the true state of their projects, they need to walk the halls and talk to their people. Many a project manager has wasted precious hours, days, and weeks wrestling with links and dependencies in a scheduling program trying to find the answers to their problems. What they should have been doing with that precious management time was using the scheduling tool to find out where to go next in their MBWA and ETEC ("management by walking around" and "eyeball to eyeball commitments"). Project management is a never-ending process of problem solving that only gets more intense during the endgame.

### ***Flawed schedules***

One source of endgame scheduling problems is a schedule that was flawed from the start. Avoiding endgame problems begins early in the project during the task of creating a work breakdown structure, a task list, and then linking dependencies into a schedule. There are many

references that give valuable advice on how to do this. These scheduling methodologies may differ in the details, but there are two key attributes of a successful schedule creation process that are universal:

- At some point in the work of refining a task list, the people doing the work must become directly involved in the identification and content of tasks for which they will be responsible.
- The project manager must get an explicit, eyeball-to-eyeball commitment from each team member on the schedule attributes of the tasks to which he or she is assigned.

If these two imperatives are not practiced religiously by the project manager during work breakdown and schedule creation, then the schedule will be built with inherent structural weaknesses, and those weaknesses will almost certainly erupt into schedule slips during the endgame.

### ***Late changes to scope of work***

Another source of endgame scheduling problems is late requirements changes to the project's scope of work that occur after initial schedule creation. As we discussed earlier in requirements management, a high tech project can expect to be peppered with change and enhancement requests during the development cycle. And at least some of these changes, although occurring disruptively late in the project, may well be vital to the success of the product. Such changes may pass through a "sifting" process described below in Issue Management. The impact of these changes, while disruptive, is recognized by management and immediately incorporated into the schedule. Other changes may be made through "sneak paths" such as a marketing product manager going directly to a developer, or a developer implementing some extra "good ideas" without passing through a consensus. These changes end up in the hidden scope of work described below.

### ***The hidden scope of work***

The other source of scope of work changes is simply the subset of real work of a project that slips unrecognized through all of the development process steps that create tasks and schedules, and which lies hidden until finally uncovered during the execution of development. The volume of this hidden work depends on a number of factors including the technical risk of the project and the quality of the work breakdown process used to develop task lists and schedules. But even the best development process is going to miss some work. There will always be some hidden tasks.

In extreme cases, there can be months or years of hidden work. In the most dramatic manifestation of this issue, this pile of hidden tasks is discovered in one fell swoop. A project schedule that was tracking its milestones more or less successfully in one week suddenly has a 6-month slip the next week. Of course, the project did not slip 6 months in one week. The extra work was always there; it just lay unrecognized until late in the project.

### ***The project manager as a scheduled development resource***

There is a reverse effect when the project manager is also scheduled to do some direct work in the project. As the project's management becomes more demanding, it is the resource that disappears, as the project manager gets more caught up in the full time job of project

management and cannot devote sufficient time to development tasks. Or, the project's management is abandoned as the project manager dives in to move the schedule along by a brute-force attack on the tasks themselves.

As a general rule, in all but the simplest of projects, the project manager should never be assigned direct work in the project. As a practical matter, the project manager may be able to do some direct work in the early stages of the project, but he/she should not be tempted to do direct work late in the project, where endgame management needs fulltime attention.

In larger projects, there are often team leaders who have part-time management duties and are also scheduled for direct work. The project should be planned so that they have little or no direct development tasks late in the project. This allows them to devote most or all of their time to management duties during the endgame, when extra management "bandwidth" is needed.

### ***Endgame schedule management***

Though we may avoid some of the endgame scheduling problems by a good work breakdown process and an orderly change mechanism, we see that there is still likely to be hidden work that will disrupt the endgame schedule. To counter this effect, the management of the project endgame schedule involves the following behavior:

- Limit the use of schedule tools to the resolution of detail for which they are most effective.
- Avoid misuse of electronic team collaboration tools
- Use MBWA to constantly re-determine the state of the project.
- Use issue management, not schedule, as the primary project control abstraction

### ***Limit the use of scheduling tools***

During a hectic endgame, don't try to maintain a complete project schedule down to the day or hour resolution of detail with the dependencies showing exactly who will be doing what when. It IS important that the project manager know this information. But a software scheduling tool is not the right tool to use for this level of detail. These tools are great for keeping such detailed schedules when you are constructing a building. You know how long it takes to excavate x cubic yards of type a soil, how long it takes to put up y square feet of sheet rock, and you know that the wiring is always installed before putting up the sheet rock. But these scheduling tools become a time sink with greatly diminished returns when a project manager tries to track a project with moderate or high risk at that level of detail. In the extreme, project managers can find themselves readjusting dependency links on tasks that have already been completed!

Instead, take your schedule down to a level of duration resolution and dependency detail that helps you maintain weekly or monthly milestones. Then use one of the best scheduling tools available for endgame scheduling, the white board. The ideal setup is a war room, a room dedicated to the project that is easily accessible to all team members (the best war room I've ever set up was on its way to the lunch room, coffee, and the rest rooms. Nobody could miss it!). When the state of the endgame warrants a micro-schedule down to the day and even hour resolution, keep that schedule here. Keep about two to three weeks of total schedule at this resolution. Use it to discuss issues with you team, either in short 10-minute daily meetings, or in

smaller groups and one-on-one's. Periodically, transfer schedule detail up to the software scheduling program.

### ***Electronic team collaboration tools***

Some of the newer network-based team collaboration tools coming on the market have virtual team rooms with white boards and other discussion aids. These tools have some of the same advantages as a real team room. There are the additional advantages of having everything in electronic format for easy archiving and rapid sharing. However, more and more of these tools are incorporating electronic task management tools that distribute task assignments to team members and collect electronic commitments from the team. Such tools are a poor replacement for one-on-one contact between project managers and their team members. These tools have an appropriate place in groups that **MUST** be spread out geographically for good business reasons. In such cases, the tools can mitigate some of the crippling effects that such geographic separation has on good team communication and management dynamics. But these tools themselves are **NOT** a good reason for separating the team members, whether the separation is truly geographical or the kind of separation that sets in when personal communication between team members (including their leaders) becomes rare.

For teams that are working in close proximity, electronic collaboration tools can be misused. Managers of close proximity teams should personally involve affected team members in the work breakdown process and should get personal, eyeball-to-eyeball commitments on the schedule and content of work to be performed. They should track the progress of tasks through personal contact. Even when a task is on schedule, there may be mitigating circumstances that threaten the schedule. These "early warnings" might well come out in a personal conversation but remain hidden if the team member just clicks the "on schedule" box of the automated response E-mail. Unfortunately, these electronic collaboration tools are allowing project managers to retire to their offices and try to manage the assignment and tracking of tasks via the Internet with team members that are within 100 feet of his or her office. Such local use of these tools should be avoided.

### ***Management By Walking Around (MBWA)***

Management By Walking Around is the most powerful tool in any manager's tool set. By observing work in progress and having short informal conversations, project managers can often uncover a problem that might otherwise lie festering until the next status meeting, review, or test. Discovering a program slip when its a few hours in duration may allow you to quickly reconfigure the immediate tasks at hand and solve the problem before it grows silently into a month-long slip or more. Often a team member is more likely to open up with you about a growing problem in an informal one-on-one that in a meeting with other peers and management present.

Develop yourself as a problem solver in the disciplines that you are managing. You don't need to know as much as your team, but you do need to know as much as possible so that you can at least be a good sounding board and facilitator in resolving a technical problem. Many a technical problem has been solved by a project manager patiently helping a team member step through a problem at a white board. Most or all of the solution may come from the team member, with the project manager acting as a catalyst or facilitator.

Be careful about overdoing it. Keep enough contact with team members to let them know that communication channels with you are wide open, but try not to be too invasive and disrupt the work in progress. Sometimes this is a fine line to walk, but one of the tricks of effective management is to find that sweet point where you are accessible enough to be present when problems are just starting and may easily be solved. Showing up now and then during an endgame to take status and issue orders is a sure recipe for disaster. One formula that works in a tough endgame is to have a daily meeting first thing in the morning. This meeting is strictly limited to 10 minutes maximum and is used to 1) coordinate the days activities on the high-resolution schedule on the white board, and 2) find out where to go during the day's MBWA to resolve problems and issues. Only extend this meeting when the entire team unanimously feels that something needs to be discussed by the whole team (an interface design or a development tool that is used by everyone). Otherwise, schedule time with people, schedule another meeting if necessary for a subset of the team, let the team go back to work, and start making the rounds to solve problems.

## **Issue management: managing endgame minutia**

### ***Issues defined***

In one sentence, developing a product is the process of moving a design space onto a requirements space until the unresolved differences between the requirements and the design -- the "delta" -- is small enough to declare the product as having sufficiently satisfied all product goals. This delta, at any point in the development project, is the difference between the desired attributes of the product and the current state of the design in progress.

Development process documentation, if it is done correctly and "animated" during the development process, captures most of this delta in the form of requirements that have not yet been validated, design that has not yet been verified, and tasks that have not yet been completed.

The remaining items of the delta not currently captured in these documents are called issues. Some of examples of issues are the following:

- Test, review, and inspection results that indicate flaws in the design ("bugs")
- Test, review, and inspection results that indicate a change in requirements is needed.
- Test, review, and inspection results that indicate a flaw in the test, review, or inspection process itself.
- Anomalies discovered during design activities other than testing, reviewing, and inspecting.
- Observations of transient bad behavior that cannot be repeated or whose design solution is not yet clear.
- Enhancement and change requests from internal (team) and external (market) sources.
- Anything else that may impede the development of the product and which has not been captured in the requirements, design, or tasks.

## ***The endgame: issue-driven management***

As the endgame progresses, the project may become issue-driven rather than schedule-driven. This means that issues are occurring so rapidly, that the schedule at the hour and day level is constantly in turmoil. This does not necessarily mean that the schedule is slipping or that the project is out of control. This does mean that issues must be managed quickly and efficiently if the project is to be kept on track. And if the schedule does start to slip, efficient issue management can at least help detect the slip early, measure it, and deliver the bad news early enough to allow the company to maneuver before it overcommits to launch and release and finds itself with few options.

As the project becomes issue-driven, so too must the project manager become issue-driven, plugging his or her daily activities into the issue stream and resolving these issues into requirements changes, design changes, scheduled tasks, or closed issues that require no action. Issues that cannot be treated right away must not be lost or forgotten. Issues must be prioritized, sometimes unilaterally by the project manager, and sometimes by consensus decision. Issue management is a very dynamic knowledge management process. And while the project manager remains the prime mover in this process, issue management tools can help considerably.

### ***Issue management tools***

A number of issue management tools are on the market. Some are in the form of bug trackers or incident trackers. Others are more elaborate tools supporting process methodologies. But an issue management tool does not have to be elaborate to be effective. A desktop database with one table, a couple of forms, and a few report formats can do the job. Even a spreadsheet with a few columns can suffice. In general, the issue management tool should have the following key attributes:

- Submitting issues should be fast and easy for all team members. The most valuable issues are often the ones captured during observation of transient behavior that cannot be easily repeated. There may be a lot going on at that moment, and if the issue submission process is laborious, it won't get done right away and valuable observation data may be lost.
- The tool must be able to make simple queries on the fields described below. Without the ability to quickly isolate issues by State or by Owner, the tool is largely worthless for issue management.

Keep the tool simple, with just enough capability to get the job done. Don't get lost in elaborate tools with diminishing returns. The project manager and the team should be managing issues, not tools.

### ***Issue format***

The issue format should contain, at a minimum, the following fields:

- A unique identifier
- A summary title describing the issue

- A description field. The larger that this field can grow, the better. Tough issues may be open for months and have pages and pages of description as more and more is learned about the problem.
- Who reported the issue originally.
- Who currently owns the issue. There is ALWAYS an owner. By default, the project manager owns all issues not owned by someone else.
- The current state of the issue. There is ALWAYS a state. The state determines what should be done next with the issue.

There are a number of other fields that can be added, such as priority, severity, design area, product type, references to requirements tags or design tags, etc.

### ***The issue management process***

Ownership and State are the key status fields that drive the issue management process. The process uses these fields to implement a state machine that helps drive the issues to closure. This state machine can be tailored for the particular development organization's needs, but in general the process works as follows:

Virtually anyone can enter a new issue at any time for almost any reason (see "what should be reported as an issue" below). A newly entered issue is assigned the state ENTERED. As much detail as is useful should be entered in the Description field.

At frequent intervals, issues are scanned and an attempt is made to change their states. This is called "sifting". There may be several different kinds of sifting. For example, the project manager should sift the ENTERED issues daily in some fast-moving situations. The project manager may sift all issues frequently to see where to go next in his or her MBWA. Some issues need a consensus decision, a review, or other multiple-person meeting. These issues should be sifted as frequently as is necessary to keep them moving through the states. Finally, near release time, there may be sifting meetings that carefully review all issues uncovered during validation testing and make consensus judgments on whether further work is necessary or not (more about this later in Release Management). In general, the sifting process tries to move the issues through states as follows:

- Some issues need more data before a decision can be made. These are moved to the UNDER INVESTIGATION state. They are assigned an owner who will spend some time investigating and report back to the project manager. Additional information about the issue is added to the Description field (with a date and investigator's initials). The issue can then be reassigned to another state.
- Some issues will be assigned the FIX state. These issues have enough information so that an owner can be assigned responsible for implementing a fix. Note that "fix" may mean fix a design, renegotiate a requirement with a customer, update a test specification, have a room wired for a new test bed area, or just about anything else that resolves the issue.
- Some issues cannot be or do not need to be at this time. These are assigned the DEFERRED. The project manager remains owner.
- Some issues just can't be thought about at this time. These are assigned the OPEN state; project manager remains owner.

- Some issues that complete their FIX state successfully are moved to the TEST state, where an appropriation verification step is done. Other fixed issues can just be closed. Note that "fix" may mean actually testing a design change, or reviewing a changed document, or inspecting the new test bed area. The person responsible for the verification step is assigned ownership.
- Issues that are successfully resolved are then CLOSED. Issues that are resolved without any active work may be assigned a separate KILLED state to make that distinction.
- And finally, almost any other state change may occur, depending upon circumstances. For example, an issue in the TEST state may fall back into the FIX or even the UNDER INVESTIGATION state if the fix is found to be flawed. KILLED issues may rise from the dead (but if they do, we will at least be able to read about the original decision process to kill it in the Description field!)

The Description field should be updated at every state change and every time that something new is learned about the issue. This is especially valuable for issues that are not easily repeated. At each occurrence, a little bit more may be learned about the state that creates the problem. This incremental information can be vital in finding a way to repeat the problem and track down the root cause. Also, if an issue resurfaces that was previously resolved, the complete decision process that led to the resolution is available, so that the entire decision process does not have to be repeated.

It is important to keep the state machine as simple as possible, while still modeling the process that is used to track and resolve issues. The entire team should be able to freely communicate about issues in terms of their states, with everyone understanding what the states mean, how issues are moved through the states to closure, and what ownership in any one state entails. Remember that, while issue management may be a project management function, issue resolution is a team and company effort. Everyone should understand the process and how they fit into it.

### ***Responsiveness and relevance***

The key to getting an organization to report problems into an issue database is to show visible progress on sifting and resolving issues. Nothing will kill an issue management system more quickly than having issues sit untouched in the ENTERED state for a week. In the project's endgame, the project manager should be constantly driving the issue management system for management data. He or she should be generating checklists on where to go next for problem solving, creating release content for the next prototype, generating tasks lists and scheduling data, and using issue statistics to see if the project is making headway or exhibiting thrashing symptoms. The team members should see issue management as a tool to help them understand what must be done when and by whom. Combined with endgame scheduling, the path toward project completion can be seen more clearly by the team. The project moves with a visible, palatable rhythm toward completion, which can be a significant morale booster during difficult endgames.

### ***What should be reported as an issue***

It is important to keep the issue database as wide open as possible in order to capture as much information about project anomalies as possible. However, this can lead to an overloaded issue

database and a sluggish, unresponsive issue management. Every organization will have to pick a cutoff point that works for them. Here are some examples of guidelines to limit issues.

- Any observed anomaly that cannot be resolved by the observer within 4 hours should be reported. Otherwise, cascading problems may lead to the observer's forgetting valuable data or forgetting the incident altogether. The "4 hours" can be extended if, for example, the observers are religiously keeping a blow-by-blow "lab notebook", a test results diary, or other data capturing tool.
- As soon an issue is embodied in a requirements document, a design document, or an assigned development task whose output will pass through a verification step, the issue can probably be closed out.

### ***Measuring "doneness"***

As project issues are captured and resolved, the issue database can generate the same kinds of statistics as do the more narrowly focused defect tracking tools. In fact, defect tracking tools can often be expanded in scope to issue tracking, which gives the benefit of built-in defect statistical analysis tools. By tracking new issue rates, resolution rates, total open issues, etc., management can determine if the project is moving toward closure, churning on a constant number of issues, or diverging from closure as the number of open issues grows. These trends can help raise the alarm early and put management attention on the problems sooner.

### ***Variations of the process***

Organizations should vary the process to suit their particular needs. There may be additional states to better track how issues are really handled. Priority and severity fields may be added to help sift and queue issues. There may be additional fields to help capture information about the development process, such as defect containment, weak process steps, etc. In some organizations, a Quality Assurance organization may manage the system. What is important is that the system must be responsive to issues. Some tools implement some of these features better than others.

### **Team behavioral issues**

The subject of team behavior is vast, and we must be careful to keep our focus on endgame issues. The theme of our discussion here will bypass issues such as problem employees and adequate functional training of team members. We will instead focus on why teams composed of well trained and reasonably comported individuals can disintegrate into a tangle of morale and productivity problems that can impact the project goals or cause the project to fail altogether.

We identify two root-cause areas of such behavioral issues: poor development process and poor leadership. We have been treating the issue of poor development process throughout this discussion on solutions. The theme of this treatment is that establishing a solid planning and process foundation during the calmer waters of early project work can help weather the endgame trials and tribulations. But we now need to discuss leadership during the endgame itself.

## **Endgame Leadership**

Effective endgame leadership can go a long way toward avoiding team morale problems. Here are some leadership guidelines for a project manager. These guidelines are appropriate for all phases of a project, but they are especially important during the endgame:

**Stay in the trenches.** A sure method of demoralizing a team that is working long hours is to see their management working "nine to five". When you are making a push to meet a milestone and your team is working evenings and weekends, stay with them. Even when you are not directly contributing to the development work (and you probably shouldn't be -- see Scheduling above), you should be making the same personal sacrifices that you are asking of your team members. You may be able to help out by being a tester, a documenter, a listening board during informal "white-board reviews", or the delivery person for pizza or take-out. But don't issue orders for the evening or weekend's activities and then leave.

**Increase MBWA.** When things get rough, don't go off into a corner and wrestle with your project scheduling program. Stay with your team. Be positive, encouraging, and keep your sense of humor. The project management mentors that still stick in my mind after 25 years were those who could "laugh in the face of death". Keep an open communication channel so that team members will bring you problems when they are minutes old instead of months old.

**Hit Milestones.** It is amazing how differently a 10-hour day is viewed by a team that is demonstrating controlled, visible progress and a team that is not. The team that is not making visible progress may be constantly badgered by an upper management that is desperately seeking progress data so that they can synchronize the rest of the company on the development schedule. Once the team demonstrates that it can hit milestones, upper management will back off to a higher level of observing schedule progress. Upper management does not want to manage your project (at their level, this is "micro-management"). When you relieve them of this task by showing control, they will leave you to your work, and a prime source of low team morale will be eliminated. In your high-resolution schedule, define with your team frequent milestones that will be visible outside of the team. You may have to "hold some feet to the fire" to make the first few milestones. But once the team sees how differently management treats them when they become predictable, they will make strong efforts to hit those milestones.

**Centralize the team during the endgame.** If your team works is geographically distributed or has a heavy telecommute component, try to centralize the team in one geographical location if at all possible during the endgame. Bring telecommuters in more often, temporarily relocate out-of-towners, or pull team members from different departments into one area. The management benefits of having the team together will likely outweigh the inconveniences. But be careful about generating morale problems from team members who were "used to working independently" or who were "promised flexibility" in their work arrangements. The best solution is, once again, planning. When you are building the geographically distributed team during the start of the project, plan to have planned centralization of the team during the endgame and at other critical times. Plan the space needed, travel and living accommodations, etc. so that team members will not be blindsided by a change in work arrangements. With all of the advances in electronic communication, virtual office spaces, and electronic team collaboration tools, "in situe" teams still has powerful advantages over geographically distributed teams.

## Launch management

As we discussed earlier in the Endgame Process section, much of the solution to endgame launch desynchronization lies in laying a good process foundation early. A key attribute of an effective development process is a set of checkpoint reviews, or "gates", that prevent the project from becoming "overextended" -- going too far in development when risk has not been sufficiently lowered and prerequisite deliverables have not been generated or are not adequate to build upon.

### ***The real process: gates with punch lists***

Well thought out development process models and guides always look better on paper than they do in practice. In the real world of product development, the boxes are never quite as square, the lines never quite as straight, and the decision processes never quite as straightforward as ticking boxes on a checklist. There is almost never a clean "go/no go" decision at any project checkpoint, but rather a "qualified go" with a list of tasks -- a "punch list" -- that need to be accomplished before the project moves too far past the checkpoint. And if the development process does not model this real-world behavior, then the punch list is likely to become informal or hidden, increasing the risk of overextending the project.

Instead, make the punch list a legitimate output of the checkpoint review, and ensure that closure on its items is built into your development process. For example, the review's checklist can have a completion status with provision for a completion date. During the review, a consensus decision can be made to allow one or more deliverables to be completed after the fact. If the project has an issue management system in place, enter the incomplete deliverables as issues with the completion date.

### ***Consensus risk management.***

Such qualified go decisions, or "controlled over-extensions", can quickly get out of control if they are not done correctly. Each gating checkpoint review is an exercise in risk management. The review must answer the question "has risk been lowered adequately enough in the concluding phase of development to warrant proceeding into the next phase of development." In order for this risk management to be effective, the gating checkpoint review must have the following attributes:

**Proper consensus.** The right people must be present at the meeting. This includes the people who can effectively present and characterize the true state of the emerging design, and the people who will be committing further resources to the project if it is allowed to proceed into the next phase. For example, if the gating checkpoint review is to make a go/no go decision on beginning preparations for pilot manufacturing production, then the manufacturing management and staff who will be responsible for providing resources and executing tasks to prepare the pilot manufacturing must be present at the review.

**Adequate information inputs.** The true state of the emerging design must be presented in terms that all decision makers can understand. Remaining risk areas in technology should be exposed and discussed in the terms that decision makers will understand. Continuing the pilot manufacturing example above, there may remain enough design risk that an additional printed circuit board turn is possible, or tooling may be affected by a change in mechanical design, or a change in the human interface may affect a process instruction. Such risks should be explicitly pointed out to the manufacturing team so that they can make decisions on how effective pilot

production will be in its mission of emulating final production to shake out manufacturing process problems. They may elect to delay pilot, or proceed on the contingency that key deliverables will be provided by certain dates.

## **Release management**

Release management is a special case of a gating checkpoint meeting. It is the process of preparing for and executing a release decision, defined as follows:

**Release decision.** A consensus decision that the functional and attribute characteristics of the design match the functional and attribute requirements of the product to the degree necessary and sufficient to provide both an adequate fitness for use by the customer (i.e., quality) and an adequate return on investment for the company.

The release decision has a greater impact than other gating checkpoint decisions because the cost of finding and fixing product problems increases by orders of magnitude once the product has been released to revenue customers. The need for consensus among engineering, manufacturing, service, and marketing that this product has an adequate fitness for use is critical. And in order to generate an effective consensus decision, as the release event approaches, a clear and detailed characterization of the current state of the design must be available to all of these groups.

### ***Process payoff***

It is at this point in development, as the release event approaches, where the application of an effective development process throughout the development cycle pays big dividends.

- If the vision and requirements have been captured and maintained, then the company is synchronized to a sufficient level of detail on what the requirements target is for this product.
- If an effective validation strategy of reviews and testing has been employed, then any differences between the requirements and the design have been observed and recorded at some point in the development process.
- If an effective issue management system has been employed, then the set of differences between the requirements and the design can be identified at any time.

It is this set of unresolved issues that becomes the focus of attention during the release decision process. It is at this point where an endgame strategy that calls out one final meeting to get consensus and sign off on the product may break down. In all but the simplest cases, a process should be followed that more closely fits the real world of compromise between the desired requirements and the resulting design. The following is an example of such a release decision process.

### ***Release decision process example***

- Before the start of final validation testing (where the design is tested point by point against its requirements), a full consensus meeting is held where the current open issues are discussed. The issue management system should be able to quickly produce such a report. Each issue should clearly call out how the design is deviating from the requirements.

- If there are any "show stoppers" -- issues that must be resolved before validation begins -- they are resolved before testing commences.
- A complete pass is made through the validation testing. This may take from days to weeks.
- Each morning, any issues discovered during the previous day's testing are "sifted" -- rated according to severity and priority -- and discussed in a short meeting among a limited consensus group; for example, the project manager, technical lead, quality assurance or testing manager, and the marketing product manager.
- For each issue, a decision is made to fix while testing, fix and restart testing, alter requirements, alter test protocols, fix in documentation only, no fix necessary, defer to a future product upgrade release, etc. If a decision has broad implications, more people may be brought into the decision. But the daily meetings should be kept small enough to move fast, while just large enough to keep the risk of a bad decision low.
- At the end of the validation cycle, the limited consensus group reviews the remaining open issues and decides whether they have a viable release candidate. If they decide that they do not have a viable candidate, then whatever necessary development is done. When this development is completed, another validation (regression or full) testing cycle is begun.
- If the limited consensus group feels they now have a viable release candidate, then another full consensus meeting is held, the results of the validation testing are discussed and the remaining open issues presented. If closure on these issues can be agreed upon with no further changes to the design that may affect validation, then the product is released.

Using such a process establishes a rhythm in the endgame process. It helps make the release date somewhat predictable by monitoring defect rates, the severity of issues, etc. It can help the group see if they are truly converging on a final product, or if the number of issues is remaining constant or increasing. In short, the process demonstrates that you have management control over the endgame. This will keep upper management managing at their level, and not yours. And this will reduce the risk of upper management stepping in and unilaterally making a premature release decision.

### ***Late changes***

Another payoff in having employed an effective development process is being able to respond to late change requests with useful information on the cost of the requested change, both in development time and in possible increase risk to the design. The goal is not to forbid late changes (they may be vital to product success), but rather to help management (especially marketing) understand the cost of each requested change in time and in risk, prioritize the changes, and schedule the ones that are agreed to.

Marketing product managers seem to be an especially prolific source of late changes. This is to be expected. As the product gets more "real", with late feedback coming in from beta customers and sales force experiments with demo units, product managers are getting a clearer idea of what is actually needed in form, fit, and function to solve the customer's problems. Keeping schedule and design control during the endgame allows you to make quick, quantitative responses to these late change requests. The responses should be quantified in the coin of the marketing realm: impact on time to market, impact on COGS, impact on other desired features, risk of side effects, etc. A product manager may ask you for a degausser on the widget because the sales force

claims they can sell 5% more product with one. It will help with the decision process if you tell him/her that adding a degausser to the widget will cost 12 days time to market, raise the COGS by \$46.75, and cause nearby CRT monitors to turn green.

### ***The penalty for no process***

If an effective development process has not been followed, then the difficulty of making good decisions on releasing the product becomes enormous. You will not be able to accurately characterize the current state of the design. If you don't know where you are, it will be difficult to predict a schedule for when you will arrive at where you are going. In the worst case of vision and requirements mismanagement, you may not even be able to characterize the product description that you are trying to achieve!

Without such measures to support an endgame rhythm, the endgame process becomes ad hoc, the schedule becomes completely opaque, and control of the endgame is lost. This raises the pressure by management to release the product "as is", without a measurable understanding of what "as is" is. And this in turn raises the risk of a premature release with serious consequences for the company.

## **Project termination**

We cited a number of obligations that need to be met to properly close a project. Most of these obligations have clear business justifications. Disputes over their completion are usually more a matter of degree rather than whether or not they should be done.

Ensuring that all customer requirements and obligations have been met is clearly justified and difficult to argue against. Meeting this obligation is a straightforward exercise in good requirements management and punch list management. Completing the internal deliverables such as documentation, training, and tools is also easily justified as setting a good foundation for the (hopefully) long production phase of the product lifecycle. And regulatory requirements have their own enforcement mechanism for the hopefully exceptional cases where regulatory requirements don't align with good quality practices.

It is capturing lessons learned that is often completely ignored or done in a manner that precludes any useful retrieval of the captured data. As Kerzner and others point out, the reason lessons learned get short shrift in project endgames is that the value returned from this effort is not as easy to measure as is the value of getting started fast on the next project. We'll point out some ways to measure this benefit as we discuss the elements of a good lessons learned process.

### ***What are Lessons Learned?***

Lessons learned are pieces of anecdotal knowledge that describe what worked and what did not work during a development project. These pieces of knowledge help future project managers avoid past mistakes. They also help guide project managers toward tools, techniques, and behavior that really did work and yielded noticeable benefit in past projects. Some lessons learned process focus only on the mistakes; the system should also capture the "wins".

## ***When to capture***

The key is to capture these anecdotes while they are still fresh in the minds of team members. The best time to capture is during the project itself. If the development process calls out "midcourse correction" meetings where the project plan itself is reviewed and re-planning done, this is a good time to capture anecdotes for the lessons learned database.

Since there is not always time during the project to reflect on and shape this information for entry into a database, there should be a mandatory project debrief meeting at the end of the project. A non-judgmental meeting process is essential here to get team members to "open up" and discuss the good, the bad, and the ugly that occurred during the project.

## ***How and what to capture***

The key to an effective lessons learned process is to capture both the anecdote and enough context so that future project managers facing the same situation can find the anecdote easily without wading through too much unrelated information. There are no hard and fast rules here about what context should be saved. But, in general, it is better to err on the side of more generality in context. This ensures that lessons learned items will be found by project managers who are in similar circumstances. If there are a large number of specific categories and key words, it may be difficult to search effectively. Some suggestions for context include the following:

- What kind of project: hardware, software, embedded systems, combinations, etc.
- What technology or technologies were predominate
- The product family or other product characteristics
- The development process steps, phases, or deliverables affected
- References to supporting documentation such as issues, bug reports, audits, reviews, etc.

The anecdotes should be wordy enough to capture what went wrong or right and give advice on how to repeat the good and avoid the bad. You may want to have two fields here: a description of the situation that occurred, and a recommendation on how to repeat the win or avoid the pitfall.

If the organization has a written development process, the lessons learned anecdotes will often point out things that are missing, need changing, or need expanding and emphasizing in the current process. There should be explicitly tagged with a context field so that they can be easily retrieved and analyzed when the development process itself is up for review and revision.

The database application should not be elaborate. Some off the shelf bug trackers and issue tracking software may suffice. If the organization desires customization for specific needs, a desktop database and a small amount of programming should be all that is required. Avoid getting lost in the tools. Simplicity and ease of use will be the key to effectively capturing lessons learned information. The point here is to simply tag some anecdotes with adequate context so that they can be reviewed at a later date by projects with a similar context.

## ***Retrieval and use***

When the effort is taken to capture and house these data in a database, then retrieval becomes a straightforward matter of querying the database with the appropriate context and generating reports. Again, it is better to err on the side of generality in these queries to be sure that useful information is not missed.

Once the database has been primed with data, a routine step in the planning of all new projects should be consulting the lessons learned database for the advice, warnings, tricks, tools, techniques, pitfalls, and recommendations that are relevant to the new project. Incorporating these data into the project plan itself is a powerful way to measurably improve the performance of the development team and the development process itself. Which leads to a vital element ignored in most lessons learned processes.

## ***Measuring value***

A lessons learned process is an investment of time and materials that will pay big dividends if it is properly maintained with routine and effective capturing of lessons learned data. However, like most investments, the return on investment comes later -- in this case, in future projects. Management may rightfully ask to see some measure of this value before fully committing to the investment effort necessary to effectively debrief teams and maintain the tools that house the information obtained. Showing value is often the area that is ignored in lessons learned processes. This lack of measurable value is the principal reason why these systems either fail to be implemented at all or atrophy into ineffectiveness by a lack of commitment to the process.

Measuring the value has some innate difficulties, such as how to identify problems that did not occur. Here are some ways to build some measures into the lessons learned process itself.

- Provide a "feedback" field that allows project managers to comment on the effectiveness of the knowledge obtained. This could be as simple as a numeric score from a default zero "never used" to a five "saved my bacon". This score might only ratchet up, showing highest single value given to date for that lessons learned item. Other feedback fields could include "number of hits", where the number of times that the advice was incorporated into a plan is counted.
- Count the number of lessons learned items that actually caused the development process of an organization to be modified to incorporate them.
- Some sage organizations routinely use project management metrics such as "lost revenue per project day", "opportunity cost per day", "burn rate per day", and other such eye-opening metrics. These organizations should record in terms of those metrics significant wins in future projects that were due to a lessons learned item. Again, this could be a cumulative field in the lessons learned record that records decreases in costs or increases in potential revenue that will be realized due to the lessons learned item.
- The periodic "mid course correction" reviews and the project debrief reviews should devote time to capturing value information on current relevant lessons learned items, as well as identify new ones generated by this project.

These data need not be precise. The point is to show in measurable terms that the lessons learned data is making a measurable impact on the business.

Note that the custodians of the lesson learned process will need to be proactive in the capturing of this value information, as well as in capturing lessons learned items themselves. Until the lessons learned system has been primed with enough data to show measurable value, it will be vulnerable to "attack" by well-meaning managers who see more tangible value in moving project labor quickly to a new project rather than invest the time in debrief meetings and database tools. But once the cycle of "we made a mistake, we remembered the mistake, we avoided the mistake in a future project, and we saved some significant money avoiding it" has been completed, any management resistance will fade. Management may well become the biggest boosters of the process.