Patterns for the End Game

Mark Prince, Andy Schneider

m@darkerbyfar.com (Syntegra)

andrew.schneider@pobox.co.uk (BJSS)

Abstract. The end game is an oft forgotten phase in a software project; despite the fact it can make or break a project. Successful project managers and technical leads seek to ensure their project follows a controlled glide path to a successful conclusion. However, things can and will go wrong and then success is characterized by an ability to adapt and change to meet and manage new challenges. Patterns for the End Game captures some of the common practices seen in software projects that are in, or near to the end game.

Introduction

This paper represents part of a larger effort to capture a set of patterns to fill the knowledge gap that confronts the inexperienced and aspiring Team Leader and Project Manager. As such the target audience for the pattern and proto-pattern work is IT Software professionals that have had between 2 and 5 years industrial experience. Line Managers and Programme Managers are not the target audience for these patterns.

The paper presents a number of patterns that are almost exclusively targeted at the last stages of software development projects, to help the reader to manage when the real project crunch arrives. Furthermore, understanding from the outset what can actually go wrong at the end will aid the reader to factor planning and management of the End Game into the beginning and entirety of project. Hopefully this will ensure potentially expensive risks are seen early and managed before they mature into project threatening issues.

Whilst all projects are different, applying these patterns should help the reader:

- Improve the efficiency of communications between geographically dispersed formal test and development teams.
- Develop a project plan that supports the production of all required deliverables.
- Estimate broadly whether a project is on track to deliver within specified constraints.
- Limit the amount of time managing metrics in a dysfunctional environment.
- Provide mechanism to test the system once it has gone live.
- Understand the broader context of why things go wrong, and manage appropriately.

The following patterns are introduced in this document:

- PLAN BACKWARDS: Producing a good plan (whether a GANNT chart, stories or other) is hard. Since more is known about the early stages of a project, the planning focus is often on these stages and non-code deliverables such as test plans and operations procedures are oft forgotten. PLAN BACKWARDS provides a mechanism to ensure the focus of planning is on the end-to-end delivery and that all deliverables required are taken into account. PLAN BACKWARDS provides a means of identifying whether DEVELOPER ON SITE, CHANGE QUEUE and IN VITRO TESTING maybe required.

- IN VITRO TESTING: Deployed systems often need validating. Testing in a live environment can cause undesirable side effects (such as invoices being issued to 3rd parties). IN VITRO TESTING describes mechanisms for the partial support of testing and validation in a production environment that avoids undesirable side effects.
- GLIDE PATH: It is important to know whether a project is on track for delivery within the specified constraints (time, cost, function). Without this data, projects can unexpectedly overrun and exceed budgetary constraints. GLIDE PATH describes the use of metrics and trending to provide broad guidelines as to whether a project is going to deliver within its constraints. GLIDE PATH utilises metrics and these can be overused, as such it is in direct tension with HEISENBERG'S MEASUREMENT PRINCIPLE.
- DEVELOPER ON SITE: When formal testing takes place in a different location from the development team and the journey time between sites is more than a few hours the interactions between the two teams can be inefficient and even confrontational. DEVELOPER ON SITE describes how to improve the efficiency of communication when these teams are separated by significant distance.
- CHANGE QUEUE: Refactoring, and re-design activities are mechanisms that, when managed correctly, continuously improve the quality of a codebase and reduce the rising cost of change on a project. However, in a complex project, these activities can invalidate deliverables provided by other teams. For example, schema name changes can invalidate data migration scripts and refactoring code can invalidate certain integration tests. This issue is of particular importance at the end of the project. CHANGE QUEUE describes a mechanism for managing the effects of reduced refactoring levels that should occur at the end of a project.
- STOP REDECORATING: Metrics are commonly used within organisation to provide notional measures of project velocity, efficiency, quality and other attributes. However, metric production can become a burden when management continue to change how they want metrics reported. STOP REDECORATING describes how separating the form of the metrics from the data can reduce this burden.
- PULL REPORTS: Management can often want daily updates on progress and the team end up spending time collating data and pushing out metrics updates. This pattern describes a mechanism for turning the push out of metric data into a 'pull' mechanism, where the management can access the data they need, when they need it.

Embedding the Patterns in the development process

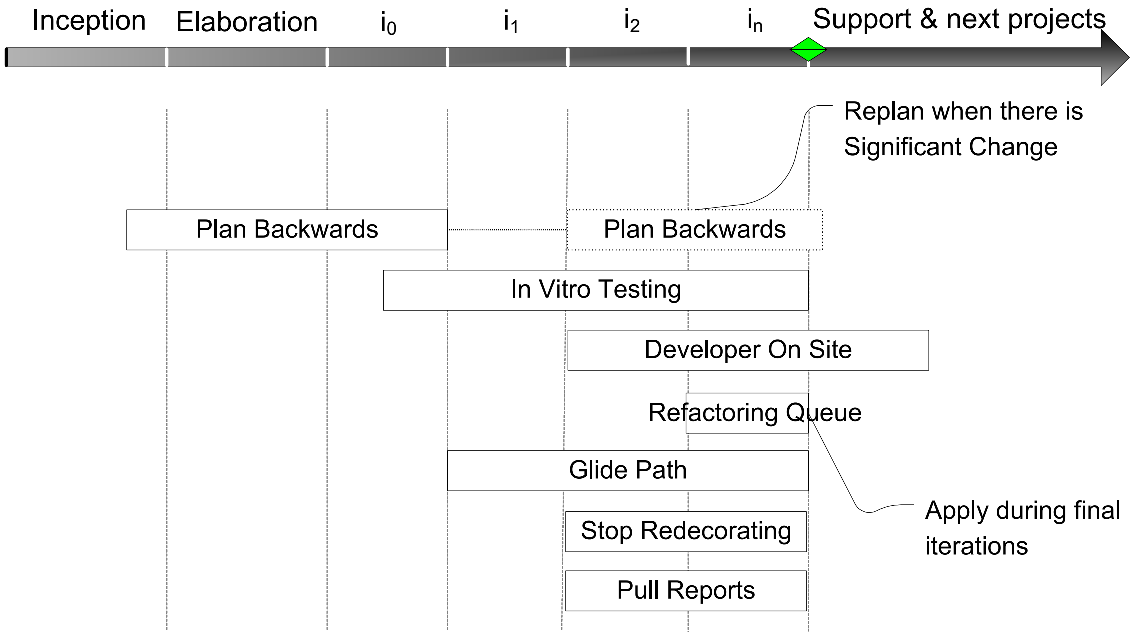


Figure 1

Figure 1 indicates when to apply the patterns described in this document. There are no hard and fast rules for this, but the diagram and following narrative act as a guide. Most of the stages along the timeline have been quoted from the Rational Unified Process [13] to utilise a common set of definitions. It should be noted that the stages described are generic to most if not all development methods. For example, XP has been mapped to RUP in [22]. The interested reader is directed to the reference for a detailed explanation of these stages. The listing below expands on the timings of pattern application outlined in Figure 1.

- PLAN BACKWARDS: Initially apply this pattern as soon as the stakeholders are engaged (during Inception when the organisation's governance body sanctions the project) and through to the end of I_0 (when all the artefacts should be understood and development is not necessarily underway). Planning backwards will enable the Lead to draw all the artefacts into the planning process. This pattern has been shown on the timeline a second time (dotted lines); it may be appropriate to reapply the pattern should there be a significant change in the project. You should not need to re-apply this if there is not significant change as without major change the shape of the project will be the same.

- DEVELOPER ON SITE: This pattern should be applied as needed (generally once development is underway and usually beyond $i_{2)}$. In most cases, a project with a new team can take two iterations before things settle down, estimates become accurate, etc. Hence, application of this pattern is shown from i_2 onwards. The responsibility lies with the team to identify when the quality of defect reporting is low and with the Lead to take appropriate action.

- CHANGE QUEUE: As the end game draws near this pattern can be applied to assist in a smooth coast to delivery without losing needed refactorings. It should therefore be applied as late as possible, normally after a code freeze or during the last iteration before formal testing.
- IN VITRO TESTING: Planning end game testing facilities into the development is an ongoing activity that should be applied throughout development. It may be the case that architectural decisions need to be made early in the project to facilitate this. If so this may be evident during PLAN BACKWARDS.
- GLIDE PATH: Apply this pattern throughout development. The Lead needs to trend when the project will complete, and how the progress of development affects the end date.
- STOP REDECORATING: This pattern can be applied throughout the project lifecycle since it only need be utilised when the Lead spots pressure on the team due to certain types of dysfunctional metric usage (see the pattern for more details). However, the dysfunctionality that drives the problem documented in this pattern is often only seen during the latter stages of the project.
- PULL REPORTS: This pattern can be applied throughout the project lifecycle since it only need be utilised when the Lead spots pressure on the team due to certain types of dysfunctional metric usage (see the pattern for more details).

PLAN BACKWARDS

Context It is the initial phase of the project and a plan needs to be developed. During this phase there is often client and/or commercial pressure to start development as soon as possible and the focus is almost entirely on developer based deliverables (such as demonstrable software). Furthermore, the end of the project can seem a long way off and the end game a vague and poorly specified affair.

Problem **There can be a tendency to leave planning the end game as a fuzzy one month task rather than really focusing about what needs to happen. The focus is on the immediate goals such as starting development. However, the immediate goals are only part of the story and unless the other deliverables are identified, focused on and planned, the project may either fail to deliver expected artefacts or not allocate enough time for them. Either way, the result is likely to be a failure to meet expectations**

Consider an example. If the project is 10 months long then you may plan out 9 months worth of development iterations. The temptation is to squeeze the remaining activities into the last month. As the end game is in the distance, any side effects of that schedule compression are not going to be felt.

What is required is a simple method for drawing attention to the end game during the planning exercise. Maintaining focus on the end game during the early stages of the project has to be a conscious decision. All too often plans provide only a cursory examination of the end game and the result is late discovered requirements and dependencies and no time in the plan to fulfil the organisations demands. However, this focus needs to balanced (do not forget the start) and needs to drive out the minimal set of activities needed to achieve a successful end game.

Therefore

Identify all the deliverables of the project (e.g. User Guides) and do not simply focus on software production. Create the end-to-end software delivery plan by working backwards from the delivery date. Ensuring that each deliverable has at least one task in the plan that makes sure it is delivered.

The Lead needs to consider exactly what is to be produced during the entirety of the project. This should not be restricted to the software but include all other ancillary artefacts, whether documentation or activity. The aim of this activity is not to produce the largest list of deliverables but the smallest list of deliverables that satisfy the needs of the project and organisation. In short, this pattern requires that the reader understand the deliverables, the reasons for them and their inter-dependencies. Once the deliverables are identified a plan can be developed that delivers only these outputs by starting with deliverables that should be on the project plan and working backwards. You might consider deployment to be the last deliverable, but really the last deliverable is support, including 3rd line development support and operational support (backups, 1st line support, etc). Working backwards, the project plan may need to factor in support of the system, deployment, quality assurance and ensure all external dependencies are in place. Once a plan is produced, managing the plan and the people involved still relies on the skill of the project manager. What this pattern provides is a mechanism to assist in the creation of the plan and not its management.

Identifying the deliverables can be difficult if there are not existing examples/standards to follow in the organisation or you do not have a great deal of end to end project delivery experience. When the authors plan they find it useful to go through a set of questions like those listed below in order to structure thoughts and tease out information. The questions have two purposes; firstly they provide a prompt as to which deliverables may be needed and secondly they can be used to help communicate the purpose of each deliverable, e.g. deliverable X will answer the following questions.

You may find it useful to build and grow a set of questions that work for you in your environment and domain. Below is an example of the sorts of questions the authors ask, but it is not an exhaustive set and the domain is medium/large business systems.

- **Support**
    - **Procedures (how-to's) and tools for operations:** Example questions to ask might be:
        - Is there a policy to archive data from the database after a number of years to keep the database size down?
        - How are defects in production replicated in support? Is a staging or support environment required?

        - Is there a clear escalation procedure?
        - How is the system backed up/restored?
        - How is system performance to be monitored ongoing?
    - **Training:** Who is producing the training material? Who is performing the training? Where is the training to be? Is there a user guide? Is it on-line or on paper?
    - **Cover:** Has support cover been arranged? Do the right teams have pagers or mobile phones? Is there an agreed rota?
    - **Business Continuity Plan:** What is the business continuity plan going to be relative to the system being built?
- **Deployment**
    - **Deployment:** How many sites? Is Automation required? Are Procedures required? Is it an upgrade or fresh install? How will we know we are ready for deployment?
    - **Data Migration**: Is data migration required from existing systems? How is the migration going to occur, is there to be a data load from flat files or will there be an extraction of data from live systems at deployment time? Will database schema need migrating?
    - **Production Environment Build:** Do you need to have the production environments built before deployment or do they exist already? Do you need a staging environment? Who configures the hardware? Who purchases the hardware?
    - **Back Out:** What happens if the deployment fails? Is there a need for a plan that describes how to back out the system in this scenario?
- **Quality Assurance**
    - **User Acceptance Testing (UAT):** How many cycles? Any dry runs planned? Where do these take place? Will you need Developer On Site? How will you know you are ready for UAT? How many cycles are needed and how much fix time will you allow per cycle?
    - **Operability Acceptance Testing (OAT):** How many cycles? Any dry runs planned? Where do these take place? Will you need Developer On Site? How will you know

  - you are ready for OAT? Is a new manager with specific Operations skills required for this testing? What is the lead-time to recruit?
  - **Deployment Dry Run:** Is the system complex enough that a dry run of the deployment needs to be performed to ensure that the organisation is ready for go-live?
  - **Quality Bar:** What is the quality bar that determines whether a system is ready for go-live?
- **External Dependencies**
  - **Departmental Scaling:** Will the new application require any extra support members? Is there budget for this? Are there skills in the organisation to support this?
  - **Service Level Agreements (SLAs):** Does the application depend on other teams or managed services for support? Can these teams meet your SLAs?
  - **Deprecated Toolsets:** Is the development team going to develop against tools that the Operations Team don't support because they are deprecated? Will the Operations team support your application even though their cover contracts with Vendors don't?[1]
  - **Co-Residence:** If your delivery is going to share production hardware with other systems, are all the library versions compatible? It may be the case that The Lead's team develops against libraries that can't be taken into production without uplifting the production versions. This could involve a significant regression test.
  - **Stakeholder Buy-in:** Does every deliverable relate back to a stakeholder requirement?
  - **Regulatory Environment:** What regulatory requirements are there on the project? How do they need to be satisfied?
  - **Environments:** How many environments will be needed? What software components will be needed in each environment? What hardware will be needed? Who will provide them, when are they needed and what is the lead time for construction? Who will manage the environments?

[1] The authors have experienced this problem when the development team were working with Oracle's Analyse profiling tool but Oracle had deprecated this tool and the operations team were using DBMS_Stats – the currently supported Oracle Profiling tool.

Once a list of deliverables is generated the existence of each one should be challenged. Can the project survive without the deliverable? Is the deliverable only for internal use within a team? If so, maybe it should only exist as a set of notes; a whiteboard within the team's oral tradition. What is the minimum amount of information needed in the deliverable to make it useful? A good approach to answering this question is to draw up the table of contents and socialise it amongst the customers of the document. Finally, consider when the deliverable needs to be produced (the later the better) and whether it can be faked at the end of a project. Whether it can be faked at the end will depend on regulatory (government or industry bodies) requirements, internal QA processes etc. Remember, whatever the deliverables are, only produce necessary and sufficient documentation. The business wants systems in a timely fashion and not documents.

Depending on the project, one or more of the above activities will need to be planned for. For instance, a small project with many releases may, after the first release, only require a limited number of deliverables, most of which will be updates. Each activity is likely to take up significant development resources. In order to plan you need requirements, and some of your requirements will surface from considering the answers to the above questions. It will be essential that you engage these teams as soon as possible in your development (e.g. OAT, deployment, Audit, Rollout) to ensure you can plan for their requirements rather than react to them when they appear late in the day. It is therefore key to ensure that the end game is as well thought out as the rest of the plan. It is important to note that these questions are not just the domain of the business system developer. Even if the development is a library for use by internal development teams, many of the above activities will still need to take place.

Incorrect Application

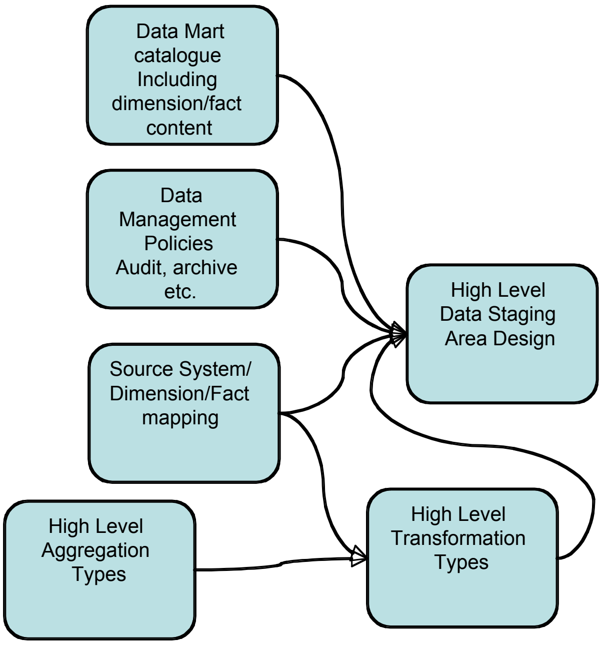
The team can suffer from analysis paralysis as they attempt to define all deliverables to a fine level of detail before proceeding with the project.

The focus on deliverables can encourage the generation of long lists of documents that have little, if any, relation to the actual end goal and may not be visible to external stakeholders (i.e. are internal artefacts). This can degenerate into a project that is a documentation effort rather than a development activity. All activities must relate directly to the successful deployment and support of the live system, anything else is wasteful and should be removed from the deliverables list.

Examples

A small project (less than $200,000 budget) produced a deliverables list at the start of the project. This list contained a thumbnail sketch of each deliverable along with any major dependencies. Planning consisted of ensuring there were milestones for each deliverable in the plan and then, working backwards, the project team generated a high level work break down structure (at a release level, not iteration level). This resulted in the production of all the artefacts.

A large project, in a 2 day workshop, produced a deliverables list by working from the end of a stage gate backwards, identifying both the deliverables and their dependencies on other deliverables. A small excerpt of the output is below:


Data Mart catalogue Including dimension/fact content
Data Management Policies Audit, archive etc.
Source System/ Dimension/Fact mapping
High Level Aggregation Types
High Level Data Staging Area Design
High Level Transformation Types

Travel Light (an XP[14] related term) is an example of the focus on generating only necessary and sufficient artefacts.

Related Patterns & Practices

Theory of Constraints: Goldratt's Theory of Constraints (TOC) [1] describes an approach to Project Management that follows a similar model to this pattern. Project stakeholders are gathered together at the start of the project to identify the intended deliverables, objectives and success criteria. Starting with the project deliverables the project plan network is constructed backwards in time by the project manager, in conjunction with experts in the various delivery areas. The aim is to ensure that the scope of the project is fully understood and planned for.

Next Game: The Next Game, Cockburn[2] explains how part of the game (project) that is currently being played must include tasks to make the team ready for the next game. This approach favours an end game first approach to planning.

Travel Light: XP[14] uses this term in relation to the philosophy of producing the smallest amount of document needed to make the project a success.

Developer On Site

Context *You are organising development support for formal testing (e.g. user acceptance testing). The testing team (more than one person) is external and is located several hours from the development team. A co-located testing team interacts closely with development. Communication and issue resolution is timely and efficient. You want to achieve that same quality of interaction with the formal test teams, despite the fact that distance separates development from the formal testing.*

Problem **When development and formal test teams are several hours journey time apart, it is hard to ensure communication between the teams facilitates rapid and efficient resolution of queries and clarifications of problems. This separation introduces latencies between information exchanges and means that communication is normally not face-to-face.**

You could dispatch a developer to the test location when an issue arises. However this introduces latency, and if the fault is transient it may not exhibit itself once the developer arrives on the scene. Whilst a tester can send a screen snapshot, observing the system when the defect occurs and then spending 5 minutes using the application post defect can allow the observer to make the technological intuitive leaps that occur when faced with a defect at the development site.

Efficient communication and co-operative working is not simply a technological matter. Since you cannot solve a non-technical issue with a technological solution alone, you cannot simply rely on remote desktop technology such as Net Meeting or VNC to achieve the same ends. Some example drawbacks of this sort of technology are:

- It is unlikely that remote desktop technology will be running all the time on a tester’s PC, and even if it is, the developer may not be observing the remote desktop or may not be in the same time zone as the tester when a defect occurs.
- Remote desktop technology may not be allowed on production client workstations.
- Remote desktop technology can be far less effective. This is due to much lower communication bandwidth and because the remote viewer is not immersed in the observed environment, rather than person-to-person interaction when problem solving and spontaneous communication and observations need to occur. See

[2] and [15].

- It is hard to observe all the ancillary events that occur in a remote environment when using remote desktop technology.

Neither of these options is as efficient as co-location, they can be acceptable if you are heavily constrained by cost but they do not come very close to the immersive environment a developer experiences when in the room with the test team.

Therefore

Place a developer on-site with the formal test team as an observer for a number of days. If you need to, fly them around the world to make it happen.

Testers can ask the developer questions and obtain immediate feedback. They can ask the on-site developer advice on filling in defect reports or for explanations about sporadic problems that are occurring. The development team can ask the on-site developer to investigate a particular issue, knowing that the right technical skills are on-site to help. The on-site developer can observe, noticing details that the tester may not and obtain a better understanding for the business scenarios being tested. There is still latency in the process, since the communication back to the remote development team is still less efficient. However, the testers now have on-site "first line support" and the developers have a trusted technical person on-site to investigate any subtle issues.

Travel costs may include £3000 business class flights and £1000 for a hotel and expenses. However, compare this with the loss of business value from a delay in delivery by one day, or the composite cost to the project of many round trips between remote sites to determine problems. Furthermore, putting developers on-site allows perceptions to be managed correctly and the cost of a poor perception of the project may be enormous.

When choosing developers to send on-site be sure they are problem solvers, understand the business domain and have good communication and social skills. In many ways the observer is part ambassador and bad impressions (caused by the difference in cultures between the customer/business/testers and developers) can cause problems later. If there are several cycles of testing, rotate development team members through the assignment. That way you expose more developers to the "sharp end" and avoid having one person stuck on-site for a protracted period of time. Exposing developers also helps them understand the end game issues more, rather than staying in their warm coding bunker.

When making this degree of commitment it is also important to ensure the roles and responsibilities are well defined. The developer needs to know there are people back at base that have the bandwidth to support him and the test team need to understand the limitations of the developers assignment, and that the developer is there to assist and not execute tests.

As well as sending the observer, make sure they have a full set of diagnostic tools. For instance, if the application is a thin client then home grown tools for saving registry settings, testing network bandwidth and latency (as simple as downloading a file of well known size from the source of the thin client) can be useful. Database inspection tools are also useful. It may be that the observer cannot plug their laptop into the client's LAN, so make sure they have their tools on a CD or USB key so they can install software temporarily.

Incorrect Application

Firstly the developer can end up helping the testing teams too much. For example, the developer may answer questions rather than teach the inquirer how to determine the answers themselves. This can cause the test team to become over reliant on the developer and the developer then becomes intrinsic to the client's testing effort. This can make it hard to bring them home if needed. Instead, ensure the developer acts in a consultative capacity. To paraphrase Weinberg [12], A consultant will tell you *how* to make jam, not stir the ladle themselves.

Secondly, if there are many testing locations then the costs can become prohibitive and less efficient. In cases such as this, it may be appropriate to use more cost effective mechanisms such as observing defect manifestation using VNC.

In contrast, this pattern can be applied when the test and development team are not far apart, say 2 hours away. In this case it can be less expensive to send a developer on-site on demand rather than pre-emptively. However, this approach is less effective if there is severe schedule pressure and every delay counts negatively towards the project outcome.

Examples

A project for a large oil trading organisation had formal acceptance testing occurring in London, Chicago and Singapore. The team deployed one developer in each location during formal testing. The developers solved a number of problems on-site and managed to reproduce and diagnose at least one significant performance issue.

A "local shore" outsource services company always sends developers on-site when user acceptance testing is in progress. This allows them to manage customer perceptions, diagnose sporadic problems and help users log defects and articulate problems.

Related Patterns & Practices

Customer On Site: XP[14] has the practice "Customer on site" which is used to optimise communication between development and the customer and ensure rapid turn around when questions or issues arise. DEVELOPER ON SITE follows the same model, only in reverse.

Local Test Environment Facsimiles: Another mechanism for diagnosing problems that are occurring remotely is to have test environments that are maintained as accurate facsimiles of the formal test environment and have the tester reproduce the defects locally by operating remotely the local system from their location.

Diagnosis Tools: Another related technique that is commonly used is to harvest all environment information possible when reporting a defect. The user executes a program that gathers data from their machine and sends this, with the defect to the defect tracking system. Many companies follow this strategy (e.g. Microsoft reporting IE errors) and this type of tool is also useful in the situation described in this pattern. It may be that you can buy a tool off the shelf, or decide it is a worthwhile investment to develop one (e.g. COTS for Microsoft platforms, roll your own for UNIX).

CHANGE QUEUE

Context You are nearing the start of the end game. The release to formal testing will be soon. The project is reasonably large and teams such as data migration and testing may find their activities perturbed by changes in the code base. Because of the inter-team dependencies here is a drive to limit change within the latter days of the project.

Problem **To keep the software in good condition the code base is continuously refactored and larger technically led changes are made. However, these changes affect other parts of the project; e.g. passed tests can be invalidated by code changes and data migration scripts may need to change.**

During development a system should be under continuous revision to ensure that software rot is kept at bay. This activity is rather like tending the garden; any young weeds should be picked as soon as possible lest small young weeds become large, spiny thistles. Development methods such as XP teach this discipline through the refactoring process. However, this activity not only includes refactorings but also more widespread strategic fixes. These may include re-structuring part of a class hierarchy or introducing a caching mechanism. In the end game there is often not enough time to cope with the risks and consequences of these changes. A trade may therefore need to be made between tactical and strategic fixes. Without this trade care must be taken when refactoring or making significant changes near the end of the project so you can prevent other teams being affected. For instance:

1. Refactoring elements of the schema by moving an attribute to a different table or changing a table name affects any schema migration activities or data load activities.
2. Changing the code base will invalidate a subset of the functional and non-functional tests. This applies to both strategic fixes and refactorings. Though refactorings are small changes (such as rename a method) that when executed 100% correctly are identity transforms (from a functional perspective), they can still introduce defects. For example, a change to an attribute in an XML schema can mean that integration tests would fail if the consuming systems were not updated. Another example would be the failure for an argument type change to be correctly rippled through the code base in a loosely typed

language. It is the chance that defects can occur that means that relevant tests have to be invalidated.

3. Many organisations who test mission critical applications (such a gateway software for stock exchanges) have highly formalised mechanisms that require the mandatory invalidation of tests when code changes and also code freezes before formal testing starts.

When the project enters the final stages, the kinds of perturbation outlined above can delay a project by days or weeks as elements are re-tested and new defects are discovered. At this stage of the game the aim is to avoid significant changes and significant refactoring activities. However, this deliberate avoidance allows young weeds to start to take root and the project is in danger of the behaviour outlined in "Broken Windows"[2][16].

Therefore

During the end game establish a change queue. Aim to keep the impact of refactorings, and indeed all non-functional changes, to a minimum. Where possible, choose tactical fixes over strategic ones and add the strategic fixes to the change queue. Action the change queue items in subsequent activities.

When approaching the project end game place a hold on any schema changes, refactorings or strategic fixes. An appropriate time to start the change queue is normally during the dry runs for formal testing. Ensure all fixes affect the minimum of code (i.e. limited in impact), even if the solution is less elegant. When you decide you need to make this trade off, start a change queue and decide how this is going to be actioned. Anyone can create a list of things to do; the key to the success of this pattern is being able to action the change queue. Techniques for achieving this are outlined later.

The queue enables the project to remember which tactical fixes will need a strategic fix later. When expedience requires a small surgical fix

[2] This reference (from Henry G. Cisneros, see http://www.luc.edu/curl/pubs/defense.shtml) describes the tendency for a building that has one broken window to quickly degenerate as it is vandalised over time. I.e. The building is fine until the first window is broken, this broken window then shows that people do not care about the building and other elements of the community respond to this by breaking further windows.

(despite the real need for a wider ranging refactoring), add the required refactoring to the change queue with a priority. There are a couple of variations to this approach.

> a) Instead of maintaining an ongoing change queue you can fake it by retrospectively creating the list of required refactorings. This works well, but there is a risk that a key refactoring is lost because it was simply forgotten during the latter stages of the project. In this case, the authors have had success using a lightweight collaborative tool for decision capture throughout the project. Usually this is a Wiki[3].
>
> b) Choose to apply the tactical fix to the maintenance branch and the strategic fix to the development code branch. Using a change queue associated with the development code branch allows the development team to select and prioritise refactorings in an orderly manner, without forgetting any. Once the changes have been made in both branches you will not wish to merge the tactical fix down into the development code branch. However, you will want to merge other functional defect fixes from the maintenance branch down to the development code branch. There are many approaches to branch management. [18] has a good collection of configuration management patterns that address branching and other topics.

Irrespective of approach, the queue needs to be stored and managed. A common approach to this is to use a defect tracking database with status Deferred (you do not want the defect to appear in the list of defects to fix in this release). Keep track of the size of the queue, both in the number of items and the length of time to fix. If the queue becomes too large then this indicates:

1. The queue was started too early.
2. The criteria for rejecting a change may need to be re-evaluated. Items on the queue need to be those that will perturb other teams when the end game is in full swing and there is limited time for the other teams to react. It is not a location to place changes and justify a ‘hack and slash’ mentality in the main code branch.
3. The product isn’t ready to go-live, since there are way too many ‘tactical’ fixes being put in to the product. During the end game the code churn should drop, not rise. Experience may suggest the development team is travelling ‘too fast’ and need to slow

[3] www.wiki.org

down. Throttling the team in this way can be achieved by pushing some of the issues through the standard defect chain since rising defects have a tendency to slow down development as management move to defect fixing.

4. The queue contents are not being actioned.

Addressing these issues and actioning items requires a strategy be settled on as soon as possible for dealing with the queue. When planning for the next phase starts, or the code freeze is lifted, there is an opportunity to address issues held in the change queue. There are two broad approaches to take. Firstly, (and often the default approach) roll up the refactoring activities in functional development, taking care to ensure this activity is not visible to management. In other words, hide the refactoring work in functional development. This technique works well when management do not understand software or refactoring and have a history of refusing time spent on any development that does directly result in new function. This model also mimics how refactoring is usually performed, i.e. as part of the act of writing code. The problem with this approach is that there may be a number of items in the queue and the time may be difficult to hide. Furthermore, this approach isn't transparent and does not allow educated discussion between stakeholders on which elements of the change queue are key.

The second approach addresses these last points by focusing on gaining agreement with the stakeholders (often just the project manager) that a certain amount of non-functional work is needed. Present a prioritised list of refactoring work with estimates and request that some of the work is factored into the plan. Obtaining this agreement can be difficult and is often met with the response "why wasn't it done right the first time?" It is rare to find management who understand that they need to invest in their existing inventory (code) to maintain productivity. Selling them a change queue can therefore be an uphill struggle. We recommend a number of approaches to negotiating content from the change queue into the project plan (as stories, tasks or otherwise):

- Apply ENOUGH FAT or THE GOOD, THE OK AND THE UGLY (both APPENDIX A) to ensure there are enough refactorings on the list that you can afford to be argued down.
- Apply the proto-pattern JAM TODAY (APPENDIX A), map refactorings to areas that are likely to be worked on in the next phase of project work. This shows the management the linkage between the technical work and the functional development.
- Apply CHANNELLING GEEK[19] which explains that it is important to explain issues to management in a language that

they will understand, that is, keep it non-technical. Explain refactoring using a simple metaphor such as: "All houses need redecoration. This project stopped redecorating to focus time on other project aspects, we now need to touch up the paintwork."

If you decide to pursue a transparent approach to the change queue then be sure to make the trade off being performed explicit to the stakeholders, i.e. inform the management that refactoring activities are being paused to optimise for the delivery date and reduce churn in dependant teams. Explain the results of this. Even when this approach is taken, it is the authors' observation that this information can be disputed or conveniently forgotten during the next planning phase. It is therefore key to have a persuasive argument at the start of the next phase of work.

Incorrect Application

Maintaining a change queue can quickly become an excuse for not doing the right thing. Instead of putting significant thought into the changes, there is a temptation to plump for the quick and easy solution and which will result in a badly thought out refactoring being added to the queue.

The change queue can also be started too early, resulting in a long queue of needed refactoring effort (for which there will not be enough time) and fragile code.

Management can see the change queue as a mechanism for saying "just hack it in, we can fix it next time around". In this case the management will be making a trade off they do not understand. It is normally always best to avoid making the change queue a document that is visible outside the team; it is just an internal document that will drive suggested work for the next planning cycle.

Examples

An agile project (25 development staff) controlled flux in the codebase once the first dry run of Use Acceptance Testing started. After that point they added desired refactorings to a Wiki and made tactical fixes that avoid invalidating tests or adding additional workload to other teams.

A smaller agile project (8 development staff) stopped refactoring software once acceptance testing started. At this point potential changes were added to a defect tracking system.

Hohmann, in [23] describes how the project team should strictly manage change in the code base in the closing stages of a project to ensure that delivery is not prejudiced by a late breaking code change.

Related Patterns & Practices

TODO: The change queue is a formalisation of placing TODO comments in code. The problem with TODO comments are:

- Different conventions can make extraction difficult.
- TODO comments tend to be somewhat cryptic, adding descriptions into a defect tracking tool appears to switch people to a more explanatory mode of writing.
- TODO comments are not prioritised until after the fact.
- There can be an enormous number of TODO comments, and sorting the wheat from the chaff can be time consuming.
- There is no external visibility to the rest of the project; the comments stay with the development team scope.

Impact Analysis controlled refactoring: During the end game refactorings are preceded by impact analysis that determines what tests are effected. When a refactoring effects a significant number of tests the refactoring is deferred. However, this needs to include not only tests but also the other teams involved.

IN VITRO TESTING

Context *The system is sufficiently complex or business critical enough that it is appropriate to validate it post-deployment. You are required to build and deploy a system that can be tested in production. You wish to provide similar test abilities to those experienced in integration testing.*

Problem **When an application is deployed it is important to be able to verify that the application is operational. The system is complex, it integrates with other parts of the enterprise and business processes can span multiple systems. Testing the application in the live system may generate undesired side effects, and side effects such as 3rd parties receiving invoices or contracts need to be avoided.**

A key problem faced by the deployment team is how to determine whether a system has been correctly deployed. If the team take a "just deploy and go" mentality then they may leave a system in production that is incorrectly configured. This problem is particularly acute in component based systems where the overall system may fail due to something as simple as a failure to deploy the correct version of the message routing configuration.

If no effort has been made to test the quality of the live system when it's deployed and before it goes live, testing it as a live system can be a problem. Consider a deal capture trading system: entering a deal into the system could result in a "contract to buy" being sent to a customer. You could turn off contract publishing, but then you are not testing the entire system, only one part. What if it is the contract publishing that is incorrectly configured? How do you determine messages that result from business actions reach the correct components?

Therefore

Provide the ability to test a system while live without generating unwanted side effects. Relating to PLAN BACKWARDS, plan to test the system "in live" ahead of time.

The following techniques are suggested, but are not exhaustive:

- Build dummy reference data into the data deployment. Tests can then reference the dummy reference data. Use DUMMY DATA FILTER (in Appendix A) to ensure that this data can only be used when in-vitro testing.

- For systems that communicate with the outside world, ensure that there are end points that route back to the host organisation. See NULL END POINTS in Appendix A.

Incorrect Application

There is a temptation to build a complex in-vitro test harness. If you find that the support for in-vitro testing extends to more than dummy reference data, some data filtering and some probes to determine that communications reached the right place then the project is becoming a test tool project rather than a functional development.

Examples

Part of the deployment of a trade capture system involved entering deals and ensuring the correct output was distributed throughout the system. The dummy counterparties allow these deals to be entered without external parties being affected. All other systems that could be affected by entered deals (exposure calculators) did not process messages against the dummy counterparties**.**

A medical system used test patients and organisations throughout the system. Test reference data was used to allow test medical data to be entered against patients and followed through the system to the end points (which where internal). Documented tests could be run at any time in the production environment to verify the system was operating successfully,

Related Patterns & Practices

GLIDE PATH

Context I*f the project is not on schedule then corrective action may need to be taken to avoid budget overruns, liquidated damages or inability to realise business benefits due to late delivery.*

Problem **You want to know if the project will finish on time but determining whether the project is on track for completion is hard without a crystal ball.**

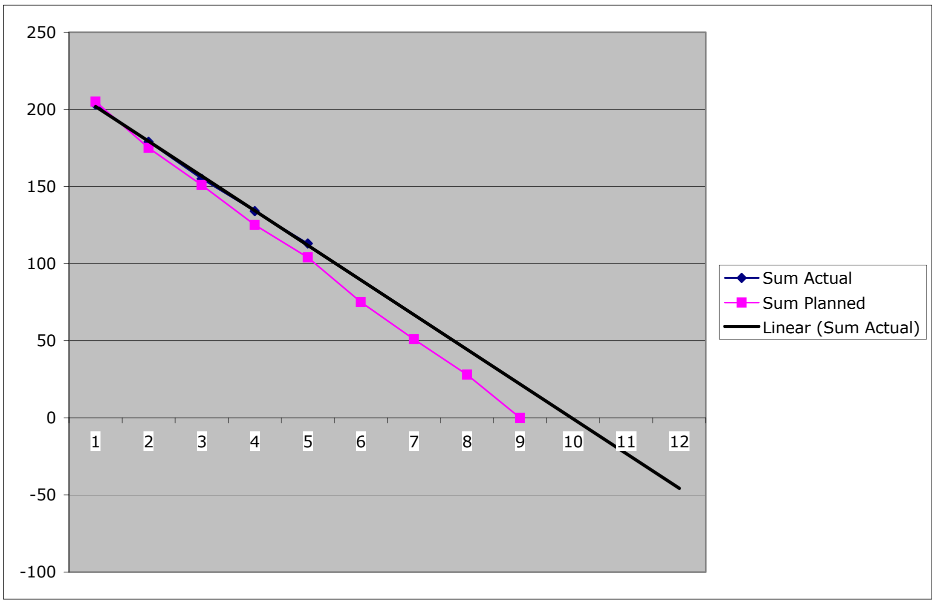
During the project you have diligently collected task completion metrics, velocity and plotted Earned Value graphs (see [24] for a brief outline of Earned Value) showing progress against plan. However, progress against plan is only one dimension. If all the tasks are complete but the quality is poor then go-live will have to be delayed. If there are only be a few defects, but testing isn't on target then the lack of defects may be misleading. Collecting this data is half the battle, but the data only tells you where the project is and not where it is going

Therefore

Trend metrics to determine when your project is going to land.

Whilst metrics do not provide an accurate estimate of a delivery date, they are the KPIs (key performance indicators). If the KPIs are significantly outside the acceptable tolerances, then this is a sign that the project may not deliver on time. This is not management by metrics, but use of metrics as project health indicators.

Earned Value Analysis or project velocity can be used to determine when the current outstanding list of tasks will be complete. Open/Close/Fix defect rates allow a measure of the quality of the software and how quickly it is stabilising (or destabilising). Complexity metrics, code churn rates and other code measures provide insight into whether the code base is stabilising and whether design complexity is rising rapidly (a bad sign). The graph below shows how the current project velocity does not deliver until period 10 rather than the required period 9.



However, this only gives part of the picture. Be sure to collect quality related metrics such as:

- Number of new defects per unit (such as time, LOC, function point).
- Number of defects fixed[4] per unit.
- Number of re-opened defects per unit.
- Defect density per component.

Collect this data on a per-severity level; this will take virtually no time at all if you use a defect tracking tool. If you do not use such a tool, you should do. You need to do this because you will care far more about the number of critical defects at go-live than you will about minor defects. Once you have collected sufficient data[5], start trending out to the 0 defect line (taking into account the defect discovery, re-open and fix rates). Determine the quality bar for each severity of defect and check that the quality bar is attainable by ensuring the intersection of the acceptable defect count and the date is earlier or the same as the agreed deadline for end of development. Figure 2 shows an excerpt from an open defect graph. Each horizontal line is a quality bar, and delivery cannot occur until there are fewer defects than are specified by the

[4] That is fixed and tested. If a fix is not tested then the defect cannot be considered fixed.

[5] 5 or 6 data points

quality bar. The red dashed line shows the approximate trend for critical defects (from Excel). The quality bar for critical defects is 0 and the current trend shows that, at this point in time, the quality bar may not be attained without increasing quality/improving the defect fix rate.

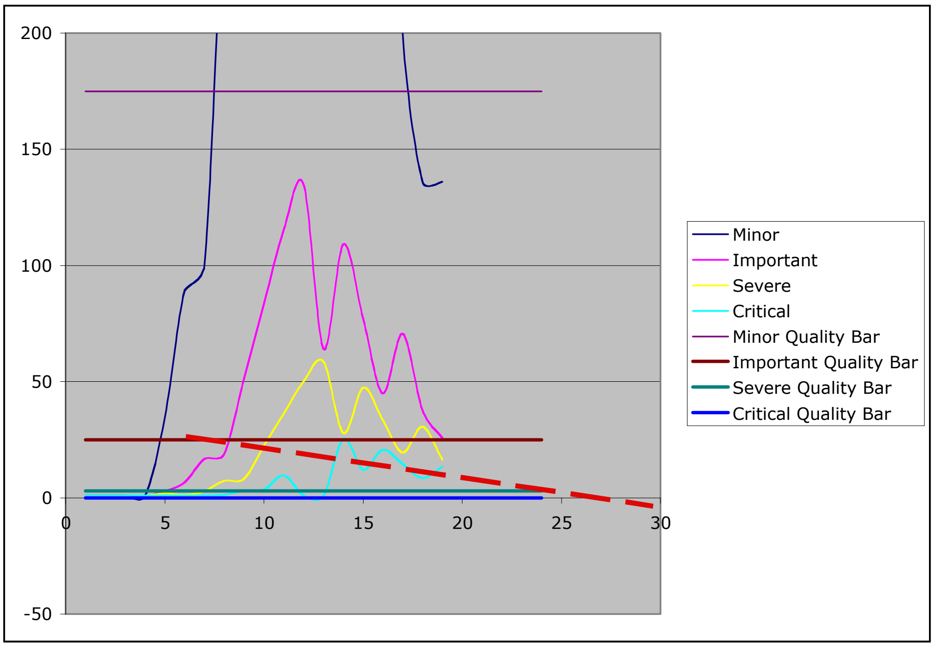


Figure 2: Trending defects

This type of trending is also useful in spotting continually shallower trend curves (quality is dropping) or trends that do no intersect the 0 defect axes (the defect rate is >= fix rate).

Do not simply rely on trending. A well-run iterative project may exhibit a saw tooth defect count (one tooth per iteration). However, come the next iteration, more defects will be generated. This is fine, as long as the overall trend tends to the appropriate quality bar. Trending also requires there be enough data for it to be statistically meaningful. When the numbers initially start to come in wait until there is sufficient data to draw conclusions, do not present findings based on the first weeks data.

The collation of progress and defect data provides two trends to guide the glide path of the project. The first shows you when quality bars are likely to be hit and the second shows you when functionality is likely to be complete.

Each discipline (development, testing, deployment, etc.) will have 2 or 3 key metrics that help identify the profile of the glide path. This pattern has mentioned two and these are not the only metrics. Others you find appropriate should also be used.

Incorrect Application

Use the data as a guide, not a master. Projects can become solely managed by metrics (see Heisenberg's Measurement Principle). Trending can increase the temptation to do so because it provides some pointers to whether the project will land in the right place and at the right time. When data and trends are considered as truth, rather than as crude indicators, the management team is inclined to optimise for metrics rather than consider the entire system. This can result in short sighted decisions that search for local minima. Similarly, just because the numbers are good does not mean the project is in good shape. The defect count may be being kept in check by too many late nights and the project team may be about to fray apart.

Examples

An $18M agile development project used metric trending to predict end dates. These predictions were not accurate but were used as indicators to progress. When trends varied significantly from expectations the root causes were analysed and corrective action is taken.

A deal capture system was being developed. The project had a number of quality bars defining how many defects of specific types could be tolerated when they system went live. The defect open and fix rates were trended to show when the project would reach a point when the quality bar was being reached. This trend showed that the project was behind schedule from a quality perspective, despite hitting the functional content milestones. Using this information, the management re-planned the project and then continued to use the data (amongst other metrics) to provide an early warning system.

Related Patterns & Practices

Big Visible Charts: [17] describes how plotting metrics on large wall charts provides an instant view of the state of the project. Glide Path is an extension of this that includes trending to provide a broad view of how the project may progress over time.

Bug Convergence: The Microsoft Solutions Framework describes in [20] the use of trending when describing how to use the Bug Convergence (fix rate exceeds discovery rate) as a mechanism for determining “that the end is actually within reach”. The same document also describes using defect metrics to identity patterns in metrics such as the Zero Bug Bounce. In this case, the Zero Bug Bounce is an indicator that the solution is starting to stabilise.

STOP REDECORATING

Context *You have to report the progress of your project to your management using metrics. Different members of the management require your metrics to be formatted in different ways.*

Problem **The metrics reporting needs of your management are constantly changing, and you find yourself wasting an ever-increasing amount of time preparing and formatting your metrics to satisfy these changes. Furthermore, management sometimes mistakenly try to measure problems away, requiring more and more formatting.**

The recipients of your reports may range from another department simply understanding what is going on in your project, through to those that have significant influence on either your project or your future. As such, they each need their reporting requirements satisfied, but tailoring these reports is becoming an increasing waste of your time. Furthermore, as your audience starts to understand the information you are presenting, they start requesting changes, which in turn causes rework and further waste of your (and possibly your teams) time.

This type of destructive behaviour reflects HEISENBERG'S UNCERTAINTY PRINCIPLE (see Appendix A)[6].

Therefore

Have your recipients apply their own formatting to the information. Analyse the current metric set, determine the purpose of the metrics and attempt to build the simplest set of metrics that achieve the same end. Design your metrics' publishing systems to be independent of the view the recipients of the information require.

The varying formatting your management will want to apply to your metrics will generally not change the actual metrics you derive. As such, be sure to provide your metrics in a presentation neutral format,

[6] Which states that whenever you measure the property of a particle, you affect its behaviour

such as a Comma Separated Value (CSV) file; be sure to include column headings. Management can read these files into a spreadsheet or database and format them as they choose. This allows you to separate the view of the metrics from the model you generate, and may remove your formatting burden altogether. In order to ease adoption, the following approaches will be of benefit:

- Provide default views that will still work with your CSV file.
- Discuss your ideas with your management and attempt to get them adopted; socializing your approach first will prepare your audience and make them more receptive.
- Read up on software metrics - this way you can quote other people's findings about what is actually useful. This is often a more persuasive argument than just asserting your beliefs.

Incorrect Application

If you assume an over zealous approach and resist providing new metrics when they are justified, or resist providing default views, you run the risk of turning your management away from your approach.

Examples

A testing team was being used as a quality-gate for the development team. Development quality was low and in order to understand where development was going wrong, management was demanding more and more metrics with varying degrees of sophistication of presentation. Generating and publishing this information became a significant drain on test team resources. Analysis revealed that a smaller set of metrics could be delivered as a CSV file. All recipients of this information were then able to manipulate it in Excel to generate the richer picture they desired.

The authors were once asked to present the same set of results in both HTML and PowerPoint. Since applying Stop Redecorating, the management were convinced that they could do all the formatting, presentation and publication they wanted themselves using Excel, Word and PowerPoint.

Multiple datasets were being used to provide metrics reports, and Stop Redecorating was applied to rationalise the reporting, and the recipients started formatting to show their own results. Further changes in metrics requirements meant that further changes needed to be made to the different CSV files that went out to the recipients. Analysing the datasets produced a single CSV file that could satisfy all reporting requirements. Column headings in the CSV files afforded a backwards compatibility between the new file and the ones it was replacing.

Related Patterns & Practices

PULL REPORTS

Context *Your project has entered a very critical phase and your metrics gathering is producing a lot of base data. You have a wide audience for the metrics, who may be in different time zones. Your audience requires your reports with a high frequency (perhaps daily).*

Problem **You find yourself struggling under the volume of data being produced, and are wasting a lot of time chasing the collection of the data, collating it and then quality assuring the resultant metrics. This loss of time results in you missing reporting deadlines.**

This problem generally surfaces near the critical End Game phases of a project, when tensions are high and all eyes are on the measurement of your final quality.

It is quite likely that the teams delivering the data towards the end of the project will deliver that data to you in different formats. Typical examples are defect rates, which will be reported through a management tool of some sort, through to formal test results, which may be paper-based. For the former, there is often a database of some sort that can be queried and reported on, but for the latter, hand collation and management results is a very error prone task.

If the burden of gathering data is causing you to miss your reporting deadlines, or if you are having to sacrifice the quality in your reports to hit your deadlines, there is a chance that your project will be perceived in a poor light, which may further increase reporting demands.

Therefore

Make the metric data manageable, and keep it centrally in a database that provides ease of processing.

In order to do this efficiently, you will need to provide the project's teams with a method of entering their data which is both easier for them to use than paper (or Word), and provide them with direct entry into your database. Typically, this can be a simple web form on the front of the database. Have the front-end capture the same data as the paper based form in order to make it as easy as possible for your users to use.

Once you have managed to automate the arrival of data into your metrics system, provide a set of canned reports that anyone can use to query your results set, and publish this set of reports (typically as web pages) to your audience. This has two positive consequences – firstly, audience are now able to PULL REPORTS, and secondly you are moving closer to a TRANSPARENT PROJECT (see Appendix A). If your management can view your project's progress at any time, they will have confidence in what you are reporting.

Incorrect Application

Your reports can escape into the wild and multiply. If you have the kind of organisation where dissemination of information requires caution, you should take measures to restrict access to your reports. The authors suggest password protection (to whichever level of sophistication you need) or something simple such as restricting reports to originating IP address. You must take the requirements of your own organisation into account.

Your database is so open that reports can be produced on the raw rather than prepared (not cooked!) data. This would make it possible for incorrect interpretations of your data to make it out to a domain over which you may have little control.

Examples

The following is a real world example from a project centralising a legacy distributed architecture in order to increase reliability, standardise operations and reduce costs.

The project was in a crucial test phase and:

- Test results were emailed in from three globally distributed test teams.
- After a couple of weeks the defect rate was actually rising. Management decided to increase the measurement periodicity. The daily effort of gathering, sorting and formatting metrics could be as high as 3 hours a day when lots of tests were failing.
- A decision was made to concentrate on getting the team clear running, rather than reporting metrics that weren't adding a lot of value on a day to day basis.
- Despite the lack of value, management were upset when daily reports weren't being delivered, labouring under the misconception that you could test quality into the deliverable.
- Pull Reports was applied in two phases. Firstly, a Microsoft SQL Server database with web forms and reports was produced (3 days effort in total). This was rolled out to all test teams; this phase completed when the test teams were entering their data exclusively into the database. For the second phase, all recipients of the metrics were given the new URLs to the metric

reports and started pulling their own data.

In the same project, management wanted to produce their own reports based on the results coming out of the web based reports. Rather than produce new reports, the project provided a VBA[7] plug-in for Excel. This installed a button on the first worksheet that pulled all the required data from the database into the worksheet. Now management could pull data whenever they want, archive and mine it as required, with no intervention required from the development or test teams.

Related Patterns & Practices

Conclusion

The end of a project is often forgotten until too late. Hopefully this paper will have shown some practices that can mitigate against this.

Appendix A. Proto Patterns

This section contains thumbnail sketches for patterns that have yet to be workshopped but are references within this text.

- **JAM TODAY**: During development many activities occur that do not directly relate to user facing functionality. For instance, code in a certain area may need tidying up, but management often do not see that effort as being directly related to the systems functionality. In effect, management always want "jam today[8]". A tactical solution to this problem is to try to tie activities directly to functional development. For example, instead of indicating that 5 man days effort is needed to improve rendering code, describe how the ABD display function will take 15 days unless we improve the rendering code, in which case it will take 9 days. This approach is easier to understand and also facilitates prioritisation. If ABD display function is not to be implemented then maybe the tidy up of the rendering code is not strictly necessary at this stage and money would be better spent elsewhere. For some activities the cost may need to be spread across a number of functional units. Another approach is to consider a defect

[7] Microsoft Visual Basic for Applications

[8] Jam Today is based on an English expression. "Jam today" refers to people's tendency to prefer the tangible relevant thing now, even if it is at the expense of longer term strategic goals.

based approach. Describe how many defects are normally fixed in a specific area and then describe quantitatively[9] how these defects may reduce or become quicker to fix if the improvement to the rendering code is undertaken. This type of approach is often referred to as "Management by Fact", an approach many management consultants take. JAM TODAY turns "Management by Fact" to your projects advantage.

- **ENOUGH FAT**: Your management always like to feel they are making a positive difference and will always try and trim down whatever work you propose. Whilst determining the minimum set of activities that need to be done is a good thing, an overzealous manager can be counter-productive. If the manager cannot be educated then this pattern can help. Always ensure the list of options or activities contains some items that obviously (though not too obviously!) can be removed. Management can then bargain you down, removing the obvious candidates, and you can ensure the key activities are still in place.
- **THE GOOD, THE OK AND THE UGLY**: When presenting management with options it is important that they actually have something to choose from. However, the manager may make what is believed as the wrong choice because they may be in a rush or do not understand the ramifications of the decision being made. To manage this dysfunctional situation always provide an ideal choice, the set of your OK choices (a choice that is good, a choice you would be happy with but isn't ideal) and then an obviously bad choice. The manager will dismiss the ideal choice and the obviously bad choice and is then left choosing between two choices that you are happy to accept.
- **TRANSPARENT PROJECT**: Ensuring a project is not a black box, that progress and quality is clear, and that everyone has access to timely information. These are key both to building trust and to avoiding micro-management from management and/or stakeholders.
- **NULL END POINTS**: For systems that communicate with the outside world, ensure that there are end points that route back to the host organisation. For example, configure a dummy counter party with an internal telex number. In this way the end-to-end system behaviour can be tested. If the system uses messaging then ensure there are tools to reconcile messages sent against messages received. In this way you can be sure that the correct messages were sent and received by the correct components.
- **DUMMY DATA FILTER**: To ensure that normal users cannot use dummy data that is used in IN-VITRO TESTING ensure the system is built to distinguish between the live and dummy data. To achieve this, dummy data must be filtered from the user interface unless a specific user is using the system or a specific preference such as 'see dummy data' is set. To distinguish dummy data from live data either pre-fix the name of the data item (e.g. short counterparty name) with a symbol such as DUMMY, add a boolean DUMMY column to the relevant data tables or use a different key range for

[9] Of course, the numbers will be a guess, but at least some degree of approximation has been attempted. A rough guess is often better than no guess at all and many managers like numbers.

dummy data. Alternatively, if you have a role based authentication model that restricts access to certain subsets of the data use that, create a special deployment role and allocate all dummy data to that role.

- **HEISENBERG'S MEASUREMENT PRINCIPLE**: The more you measure a software process, the more you may slow it down. When a project is in trouble one of the anti-patterns that occurs is the inclination to measure elements of the projects process in more and more detail. As analysis of the metrics leads to little enlightenment and problems persist, the management, rather than acting in an analytical manner, resort to requesting more measurements. Consequently the project spends more time collating and distributing metrics, making a bad problem worse. A project may not be able to stop this activity occurring, but STOP REDECORATING and PULL REPORTS help control the issue.

References

[1] Jacob, Dee Bradbury & McClelland, William T. *Theory of Constraints Project Management*. 2001 The Goldratt Institute.

[2] Cockburn, A. *Agile Software Development*. Addison Wesley 2002.

[3] Poppendick, Mark & Poppendick, Tom. *Lean Software Development*, Addison Wesley 2003.

[4] Walker, Royce. *Software Project Management, A Unified Framework*. Addison Wesley 1998.

[5] O'Connel, Fergus. *How To Run Successful Projects II*. Prentice Hall 1996.

[6] Anderson, David J. *Agile Management For Software Engineering*. Prentice Hall 2004.

[7] *Harvard Business Review on Decision Making*. Harvard Business School Press 2001.

[8] Huczynski, Andrzej & Buchanan, David. *Organizational Behaviour, An Introductory Text*. Prentice Hall 2001.

[9] DeMarco, Tom & Lister, Timothy. *Peopleware*. Dorset House 1987.

[10] Cockburn, A. *Surviving Object Orientated Projects*. Addison Wesley 1998.

[11] Brooks, Frederick P. *The Mythical Month*. Addison Wesley 1995.

[12] Weinberg, J. *The Secrets of Consulting*, Dorset House.http://www.xprogramming.com/xpmag/BigVisibleCharts.htm

[13] http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/tp151.pdf

[14] Beck, K. XP, Embracing Change. Addison Wesley.

[15] Constantine, L., *The Peopleware Papers*, Prentice Hall 2001.

[16] Hunt, A., D. Thomas., *The Pragmatic Programmer*, Addison-Wesley 1999

[17] http://www.xprogramming.com/xpmag/BigVisibleCharts.htm

[18] Appleton, B & S. Berczuk, Software *Configuration Management Patterns: Effective Teamwork, Practical Integration,* Addison-Wesley 2002

[19] Patterns for Radical Leadership

[20] Microsoft, *Microsoft Solutions Framework Process Model v3.1 (http://www.microsoft.com/technet/itsolutions/techguide/msf/msfpm31.mspx)*, Microsoft.

[21] Brooks, F.P. (1987). *No silver bullet*. Essence and accidents of software engineering. IEEE Computer, 20, 4, 1019 (April).

[22] Martin, Robert C., RUP vs. XP. 2001.

[23] Hohmann, Luke., *Beyond Software Architecture, Addison-Wesley 2003*

[24] *Earned Value Management Part One,* http://www.projectmagazine.com/nov00/evm1.html.