Change Control
And
Release Management

gary.mohan@plainprocess.com

This eBook can be downloaded for free at:
http://www.plainprocess.com/change.html

Table of Contents

Introduction

Who is this for?

This is for IT professionals working in environments lacking basic change control and release management processes.

What does it cover?

Concisely, it covers the basics of change control and release management in both software project and maintenance environments.

Why bother reading it?

The few books that discuss change control and release management treat them as separate topics. One needs to be understood alongside the other. In some organizations, the people responsible for change control and release management are separate individuals, in a management structure that creates conflict. Concisely, this book explains the topics together.

Change Control

The purpose of introducing change control is to ensure that changes in IT systems are made with reference to business priorities, urgency, scope, budgets, deadlines, risk and feasibility. The consequences of failing to manage changes can be quite serious, with project overruns, system instability and user anger at the IT department.

Change control has three basic steps:

- Logging CRs (Change Requests).
- Evaluating and Authorizing CRs.
- Planning and Implementing CRs.

Logging a Change Request (CR)

Typically, the information needed to log a CR (Change Request) is:

- Change number.
- Version number.
- Title.
- Summary description.
- List of attached documents.
- Comments.
- Urgency.
- Importance (from the business' perspective).
- Person raising the CR.
- Internal assignee designated to evaluate the CR.
- Approval or rejection status.
- Date created.
- Date due.
- Target release.

The purpose of assigning a unique change number to each CR (e.g., CR019) is to help filter out duplicates and changes with similar titles. The documentation approaches used to record CRs fall roughly into three categories; manual, semi-automated and fully automated.

Under a manual approach, CRs are logged as word processor documents, with a designated change manager maintaining a manual repository of changes. Typically, these changes will be indexed using spreadsheets. This approach works well for smaller scale environments, however, it quickly becomes unwieldy for medium and larger scale environments.

A semi-automated approach involves using a wiki or an issue tracking system to log CRs, providing consistency and an audit trail for each CR. Existing CRs are easier to search and can be grouped into categories (e.g., a wiki category for each release). Designated business users can be given direct access. This approach is preferable to a manual approach, since it is more consistent, more transparent, and less bureaucratic.

A fully automated approach involves using a tightly integrated project management suite that brings together change recording, bug reporting, project workflow and source control in a unified system. These systems are used in larger, enterprise-scale environments and allow each CR to be related to detailed source code changes. A comparison of issue tracking systems is on Wikipedia:

http://en.wikipedia.org/wiki/Comparison_of_issue_tracking_systems

Evaluating and Authorizing a Change Request

Evaluating CRs filters out bad, incomplete and duplicate requirements. The process differs, depending on whether CRs are being logged for projects or maintenance.

In projects, CRs are usually a nuisance and a source of scope “creep”, causing budgets and deadlines to overrun. Accepting CRs on a project depends on how much budget is left, whether appropriate people are available to work on the change, the impact on deadlines and whether the CR modifies the project’s critical path. Large numbers of CRs on a project indicate imminent project failure, especially if CRs increase with each user acceptance testing cycle.

In maintenance, CRs are normal. Here, the emphasis is on the level of available budget and the priority order in which the business wants the CRs worked on.

In both project and maintenance environments, the criteria used to evaluate CRs include:

- Whether bug/error reports are disguised as CRs. If the system is not working as specified, a bug report should be logged.
- Whether the change has acceptance criteria. If the business does not know how to accept the change, it does not have a requirement.
- Validation against the requirements of the system as a whole (sometimes called “impact analysis”). Changes that duplicate existing functionality should be rejected.
- Changes that degrade system quality and usability are suspect.
- Conflict with agreed future changes that this CR contradicts.

- Feasibility of implementing the change, especially if it requires radical re-implementation of the system architecture or enterprise data model.

Often, feasibility and budget limitations are the same issue. For example, users may dislike a system's user interface, however, if it is expensive to change and enhancements will not improve operational efficiency, there is no business case for the CR. Similarly, they may want a new management report, however, if this involves an expensive and radical modification of the enterprise data model, the budget required might be too much.

Knowing the relative importance that the business attaches to individual changes will help later, when planning how CRs will be implemented. A typical model for prioritizing CRs is:

- Critical: Vital for business continuity, needing urgent implementation. Typically, these are fundamental errors in business analysis, where requirements modeled in the system are so far from user expectations that the system is unusable.
- High: Important for business continuity, especially if tied to strategic business plans. Often, these will include significant changes to the enterprise data model and business logic.
- Medium: Will improve operational efficiency or effectiveness. These will be lower-level changes in business logic and user interface presentation.
- Low: Cosmetic changes, not important for either strategic or operational business reasons (e.g., removing functionality no longer in use).

Business importance should be balanced against urgency. This should avoid the classic mistake of over-emphasizing urgency above importance. A change can be authorized at the point where it is properly documented, considered feasible and assigned budget.

Planning and Implementing Changes

Assuming that a list of CRs has been authorized, implementation should be planned. The first stage of planning CRs is usually to group them together. This can be on the basis of functional area, business priority, resource availability and intended release date.

Once organized, a particular CR group may be considered so large that it needs to be taken out as a separate project. Grouping by functional area has the advantage that if the same small team work on a CR group, this may be more efficient, avoiding conflict with other teams.

Identify:

- Architecture changes.
- Data model schema changes.
- Data conversion scripts required.
- Existing source code modules to be changed.
- New source code modules required.
- User interface changes.
- Test script, test code and test data changes (especially if new test data is needed directly from users).
- Changes in interaction with third party components (e.g., a rules engine).
- Updates to third party components and associated licensing implications.
- Changes in security rules (e.g., new firewall rules).
- Obsolete code and third party components to be removed.
- Changes in the way system administrators will support the live system (e.g., new logging).
- Requirements and architecture documents to be updated.
- Performance and stability implications of making the change.

Once this detail has been identified, a task list for the work can be made, with estimates for each task. This process can help indicate early which changes will be more expensive than anticipated or represent bad requirements.

Release Management

Most books that discuss change control either do not discuss release management or treat it as a completely separate issue. They are intimately linked and it is difficult to understand one without the other. Release management is effectively the "next level up" of change control and involves three basic activities:

- Release planning.
- Determining release readiness.
- Release packaging and deployment.

Release Planning

Release planning varies depending on whether the environment is project or maintenance based. For projects, there is a distinction between Agile and Waterfall approaches. Under Agile, each sprint produces a release candidate, subject to some level of user acceptance testing. Ultimately, release management is concerned with the system build produced for the final sprint. Interim builds still matter, in terms of determining whether the final release will be ready on time.

In maintenance, release planning is driven by recurring business deadlines and any end-of-year moratorium, where changes are not allowed due to annual reporting purposes or the like.

Planning a release involves requirements collation, reconciliation against budgets and risk analysis:

- Requirements collation
  - Which individual CRs and CR bundles are scheduled for this release?
  - If larger projects are being implemented, should they have a separate release?
  - When examining aggregate requirements, can contradictory requirements be identified?
  - Are some requirements on the project critical path? If their implementation fails, what else is blocked? Can a critical path requirement be removed easily?
  - Do certain requirements depend on one another in such a way that they need to be tested in the same system build?
- Reconciliation against budgets
  - Would it be better to place individual CRs in a unified priority list rather than relying on the priorities noted for each particular CR?
  - How does the overall release look from a budget perspective?

    - In aggregate, has a reasonable contingency been allowed?
    - If the release is late or under scope, will this have indirect budget consequences (e.g., penalties with third parties)?
    - Is too much budget being burned on early releases?
- Risk Analysis
    - Business Risk
        - Is there time-to-market pressure? Does the business unit have any contingency plan, if the release is delivered late or below scope?
        - Does this release involve a business unit with a known history of specifying bad requirements, failing to support project work and failing to carry out acceptance testing within acceptable timeframes? How was this risk mitigated in the past? Will this work now?
        - Will the business unit be ready?
        - Is scope "creep" happening during project work? Are project managers regularly reporting variances on deadlines and budgets?
    - Technical Risk
        - Has the proposed work been reviewed for technical risk?
        - Was the lead architect been included in the discussion?
        - Are too many technically risky changes taking place at the same time?

The actual release plan document can simply be a list of requirements to be delivered for a particular release. It may help to add an outline of the budget for the release, broken down on a per-requirement basis.

Release Readiness

The objective of monitoring release readiness is to determine if a release will not be ready on time, on budget and within scope. During the analysis and development phase, it is possible to monitor:

- Whether functional specification documents have actually been written.
- Source control activity, seeing which changes have actually been checked in.
- Detecting which changes have bad requirements, from excessive numbers of updates to particular requirement documents, excessive numbers of meetings held for particular changes and talking to business analysts working on them.

In the test phase, it is possible to monitor these metrics:

- Results of functional tests.
- Results of performance tests.
- Defect trends (number of defects reducing or increasing).

- Source control activity related to test failures.
- The time gap between builds being placed into the test environment.

In most environments, periodic candidate builds are made available for testing. For each build, the number of functional and performance test failures can be recorded. On a build-to-build basis, trends can be detected where the number of test failures is increasing, decreasing or approximately flat. A build that has an increased number of failures is unlikely to be suitable for release.

At a lower level, it is possible to examine whether particular CRs, projects or even developers are contributing disproportionately to failures. These trends indicate which CRs and projects may need to be removed. Discussion with the managers leading this work will help to determine whether delivery for this release is feasible.

Towards the end of testing, release is more likely if each build has a lower number of defects than the previous build and failures identified earlier do not repeat themselves in later builds. It is also important to look at the speed with which failures are cleared. If each build takes longer to place into testing (compared to the previous build) release may be infeasible despite a decreasing number of failures.

Periodic readiness reports are produced, informing users and management about the status of a particular build. The information in the report is something like:

- Date of report.
- Scheduled release date.
- Current build number.
- Outline summary of deliverables included in this release.
- Per-build metric trends leading up to this build.
- Defects in this release
    - Open failures.
    - New failures.
    - Closed failures.

Ideally, the final build has zero failures. Often, a decision needs to be taken on whether to de-scope the release or tolerate late delivery. In more bureaucratic organizations, it may help to draft a formal “sign-off” document to note that a build has been accepted for deployment. If needed, the document should list:

- System name.
- Build number.
- Acceptance of listed functional specification documents.
- Acceptance of listed test plans and test results.
- That the build is complete and functionally acceptable.
- Deployment into the live environment is accepted and can proceed.

Packaging and Deployment

Once “sign off” has taken place, emphasis moves to packaging the release for deployment. In some situations, this is quite complex and requires care.

Typically, packaging the application involves these assets:

- A user interface binary build.
- Middleware / application server binary builds.
- Database schema changes and data conversion scripts.
- Configuration packages for third party products (e.g., a rules engine).
- Installation instructions, especially which items are to be installed in which order. A list of backups to be taken prior to installation should be clearly listed in this document, before the actual installation instructions.
- Identifying support resources needed for the release process.
- User documentation changes and release notes.
- A back-out plan, describing how to reverse the release should it fail.

In some organizations, system administrators will want a list of changes in application logging behavior new to this release. This is so that they can change settings in third party tools used to automatically monitor logs and raise alerts. If it is not possible to write a back-out plan for a release failing, business users should be informed of this and risks described clearly. If redundant assets need to be removed, system administrators may want this explained.

Some organizations also require that the entire release process take place in a staging environment to allow for final “sanity checking” before releasing into the live environment. The installation process may also need to be repeated into various training and failover servers. Each of these releases should be taken seriously as a full live release and not treated as “practice”. Release instructions should be practiced in other environments, prior to giving the release to system administrators.

The Business Case

Surprisingly, many organizations need an explicit business case for introducing change control and release management, despite the consequences of not having these functions in place. Attempting to control these processes can appear as obstructing responsiveness to business users, especially where the organization is small or operates on a "command and control" metaphor (management perceiving themselves as being "in charge").

The business case for managing change control and release management can be expressed in these terms:

- A consistent approach to logging, evaluating and implementing changes enhances the quality of each individual change. This has a virtuous effect, where overall system quality improves. Change control ultimately strengthens alignment between IT and business users.
- Each change is validated against the requirements of the system as a whole. Bad and contradictory requirements are more likely to be filtered out.
- Risky changes will be stopped, improving operational stability, saving money on system administration costs and enhancing user perceptions of the live system.
- Costly changes will be given adequate consideration before proceeding, saving money.
- More stakeholders are consulted about changes, enhancing analysis, implementation and testing.
- Architects are able to approach architectural changes in a more structured manner, strengthening implementation. Repeated "tactical" implementations are avoided, enhancing stability, performance and security.
- The repository of CRs acts as an audit trail for the future. This matters if a particular business unit repeatedly raises the same bad CR. Poorly trained staff in the business unit may not understand why the system functions as specified. Often, a CR is the only document available explaining particular functionality.
- Testing can be approached in a more rational, structured manner.
- Resource and deadline issues can be managed together, in aggregate, as opposed to on an ad-hoc, chaotic basis.