|  |
| --- |
| ALM for Skånetrafiken |
| Application Lifecycle Management White Paper |

PROPRIETARY AND CONFIDENTIAL

The information contained in this document is legally privileged and confidential to CGI and to the receiving party. This document cannot be reproduced in any form or by any mechanical or electronic means, including electronic archive systems, without the written approval of CGI. The receiving party is exempt from this restriction for evaluation purposes only.

If you have received this document by mistake, note that the reading, the reproduction or the distribution of this document is strictly forbidden. You are hereby requested to inform immediately and to return this document by certified mail.

Table of contents

[1 Overview 1](#_Toc437308038)

[1.1 Identity of roles in TFS & ALM 3](#_Toc437308039)

[1.2 Team Foundation Administrator 4](#_Toc437308040)

[1.3 ALM Phases 5](#_Toc437308041)

[2 Branch Management 6](#_Toc437308042)

[2.1 Branch by Release 6](#_Toc437308043)

[2.2 Branch by Feature 7](#_Toc437308044)

[2.3 Branching for maintenance 8](#_Toc437308045)

[2.4 Service Center Case Handling 9](#_Toc437308046)

[3 Versioning 10](#_Toc437308047)

[3.1 Versioning Control 10](#_Toc437308048)

[3.2 Branching 10](#_Toc437308049)

[4 ALM Artifacts 11](#_Toc437308050)

[4.1 Product Backlog Item 11](#_Toc437308051)

[4.2 Kanban Board 11](#_Toc437308052)

[4.3 Work Task 11](#_Toc437308053)

[4.4 Test Case 11](#_Toc437308054)

[5 Delivery Procedures 12](#_Toc437308055)

[5.1 CRM Delivery Procedures 12](#_Toc437308056)

[5.2 WCF Delivery Procedures 12](#_Toc437308057)

[5.3 Database Delivery Procedures 12](#_Toc437308058)

[5.4 SQL Server Reporting Services Delivery Procedures 12](#_Toc437308059)

# Overview

Due to the increasing demand in parallelism at Skånetrafiken systems integration project there’s a need in establishing a common understanding based on deliveries, branching strategies and how we commonly handle incoming support cases (Service Center, Team Foundation Server and ALM) based upon different levels of priority and severity. Another experience in the development of Skånetrafiken integrated solution is that the development organization has experienced notorious difficulty in managing changes across multiple lines of development that can occur concurrently. By supporting the previous release, while stabilizing the next release and then also performing early work on feature releases, can end up having trouble keeping each of those parallel lines of development organized. Integrated changes made between those releases is especially time-consuming and error-prone especially if developers are manually applying fixes to each of those code lines, such as a CRM-plugins, matching interface methods between services across BizTalk services in between or by keeping up with the difference in data types between the technologies representing the integration of these solutions.

Some teams at Skånetrafiken have difficulty providing information and communicating across different functional groups. Testers may not feel that they have enough information about bugs returned to them as rejected and re-opened while developers may feel that they don’t have enough information about the requirements they’re supposed to implement or the bugs they’re supposed to fix. Stakeholders/System owners together with end-users may not feel that they’ve got an easy way to provide feedback about the development progress or when they get a new system version upon quality assurance testing phases. A common question that comes up about whether an IT maintenance or a system engineering organization should own the management of Team Foundation Server environment. There are pros and cons to sole ownership by either team. IT maintenance organizations have experience in managing corporate applications, but might not fully understand the needs of software development. While engineering departments might not have the experience or expertise to manage hardware and disaster-recovery concerns that keep servers running in optimal conditions. Therefore a shared responsibility approach is always proven to be a successful way to maintain Team Foundation Server.

This document brings up those needs and suggests straight forward solutions to meet these needs by enlightening how Team Foundation together with Application Lifecycle Management is able to provide the ability for flexible approaches on fundamental level services such as versioning control (VC), work item and bug item tracking (WIT), test management, data warehouse for reporting, fully customizable and extendable framework that’ll meet these demands required on delivery processes yet to be implemented.

The complexity of the system environment consisting of *E-commerce*, *BizTalk*, *Windows Communication Foundation* and *Customer Relationship Management* is and will continue to be implemented with rapid and Agile application/system development and should also consider following the phases specified below:

* Applications can comprise multiple features
* A feature may comprise multiple components
* A feature may depend upon other features (at component level)
* A feature may have dependent features (at component level)
* Managing the process of moving components of an application between environments in a simple, repeatable and efficient manner
* Planning the release calendar of the packages, planning and mitigating changes
* Provisioning development, test, acceptance and production environments
* Version management process of specific environments
* Planning and managing development, service pack, and hotfix branches
* Quality assurance for the entire project lifecycle (quality testing, code reviews, check-in reviews, code change analysis)

The ALM for the operations phase covers the following topics:

* Deployment
* Versioning
* Monitoring
* Updating, maintenance
* Upgrades and migrations

## Identity of roles in TFS & ALM

One essential step while defining processes is also the identification of the different teams in a company that would be immediately effected by using Application Lifecycle Management.

* Developers
* Testers and Quality Assurance
* Product- and System Managers
* Project Managers
* Business Analysts
* Designers
* User Experience Experts
* Change Management and Maintenance Teams
* Release Management
* Technical Documentation and User Education teams
* Technical Support
* Information Technology Department
* Board of Executives and Stakeholders
* Business- and End Users
* Remote Teams

## Team Foundation Administrator

In order to coordinate processes and configurations monitoring the health of team foundation and application lifecycle management there’s a need in the following responsibilities by a team foundation administrator.

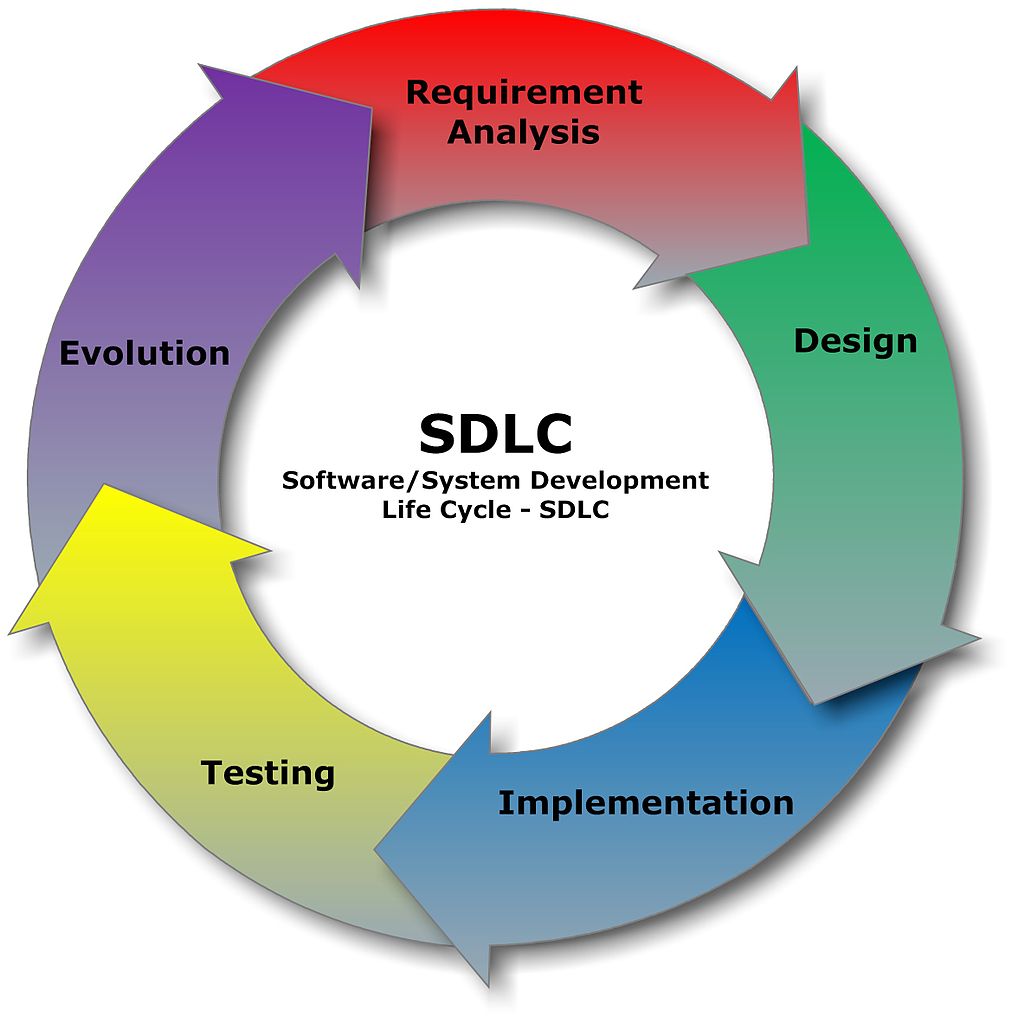
* Champion and lead the adoption in the organization
* Implement process changes
* Identify and write new reports
* Manage permissions and administer the environment
* Identify and implement maintenance windows
* Design and implement branching and merging strategies
* Architect build resources and build processes for use in the organization
* Administer the virtual lab management assets

## ALM Phases

In order to adapt ALM within a large organization it would need to be covered in different phases. It is therefore recommended that the company starts in the following order so that the complete ALM cycle is covered and fully functional:

* **Phase I:** Version Control
* **Phase II:** Work Item Tracking
* **Phase III:** Automated Builds
* **Phase IV:** Test Case Management
* **Phase V:** Reporting
* **Phase VI:** Virtual Environments and Lab Management

Currently at Skånetrafiken phases I, II and IV are almost completely functional while phases III, V and VI need either large improvement or currently are nonexistent.



# Environments



The product lifecycle is processed through several delivery stages prior production environment. Those stages are displayed in the image above.

DEVELOPMENT: During the development phase Developers are implementing change requests and incidents based on input (e.g. Microsoft Feedback Client, E-mail, Service Center or Requirements Specifications) from the Product Owner. Incidents and Change Requests are channeled from Service Center through a proxy role that is responsible in the first phase of the internal priority filtering. The Development Team is responsible in evaluating these requests based on *priority*, *severity* and time estimation.  
(Currently is it recommended that both internal CGI Developers and external partners co-exist within the same environment.)

TESTING: Based on the delivery plan the testing phase is when the product will be tested internally by the Development team (that includes Developers and Testers equally) until the product is shipped further to the Acceptance environment in accordance to delivery plan.

ACCEPTANCE: During the acceptance phase the Product Owner’s business team is providing feedback on deliverables that either pass or not the acceptance criteria and accepts deliverables further for the production environment. It is therefore recommended that the business team is given at least a week of notice prior the acceptance phase in order to perform proper testing and feedback to the developers. The deliverables are backed up with Test Cases.

PRODUCTION: In accordance to the business organization, the Product Owner decides when a delivery is due or skipped the deployment plan. It is crucial that several delivery dates are decided in advance in agreement with the development team organization to enable a continuous delivery plan.

It is also worth mentioning that during the entire product lifecycle the earlier bugs/failed test cases are discovered the easier it will become to provide fixes thus it’ll be easier to eliminate extended costs for re-delivery plans and system downtimes for these hotfixes.

# Branch Management

Branching management enables parallelism in coding, providing necessary continuity in the deliveries to meet the requirements set forth by the release planning.

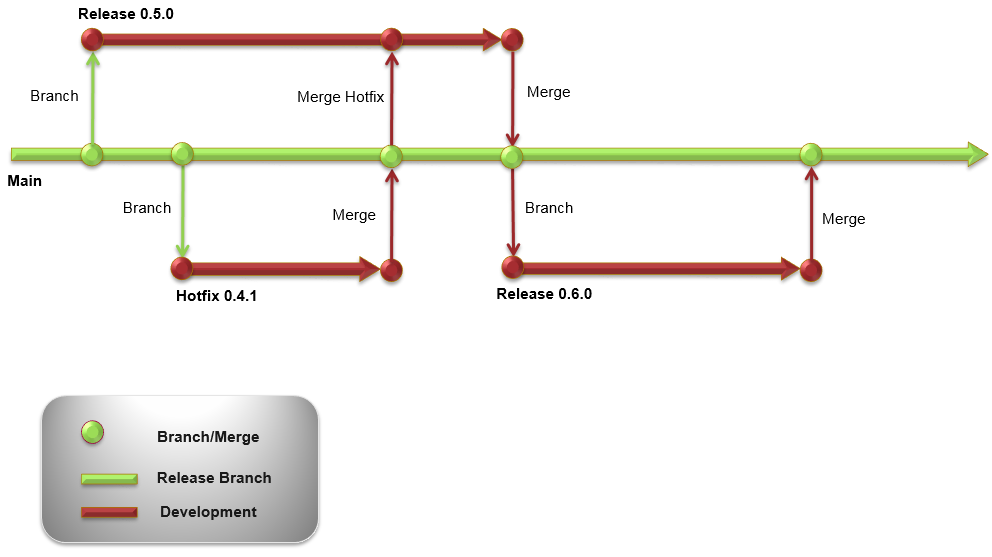
To meet these requirements it is essential to communicate and plan thoroughly release dates to avoid scenarios of complex code merging, jeopardizing loss of quality in production environment or loss of features developed by the teams.

## Branch by Release

The most common branching method is “*Branch by Release”*. Developers are working towards a common release date (stabilizing a release-branch) which at some point is merged back towards the main branch. Immediately after a new branch is created to mark the new release and the process continues likewise for future releases.

In the example below a new branch is created from main (parent branch) and is named to “Release 0.5.0”. During mid-development a hotfix needs to be pushed out. A parallel branch is created and called “Hotfix 0.4.1” to correct a bug that exists already in production environment. When the hotfix is delivered, the changes are merged firstly to the main branch and the ongoing development branch 0.5.0 at the same time.

When Release 0.5.0 is finally delivered a new branch is created immediately after so that the development process can continue unabruptedly towards the next planned release.



Even though *Branch by Release* would support parallelism in most cases, it requires that the development team is working towards the same release dates and maintenance team to support the very same features.

Having multiple development- and maintenance teams that develop both towards different or same features plus supporting multiple release dates would require a more advanced branching methodology which is explained closer in the next session of this document.

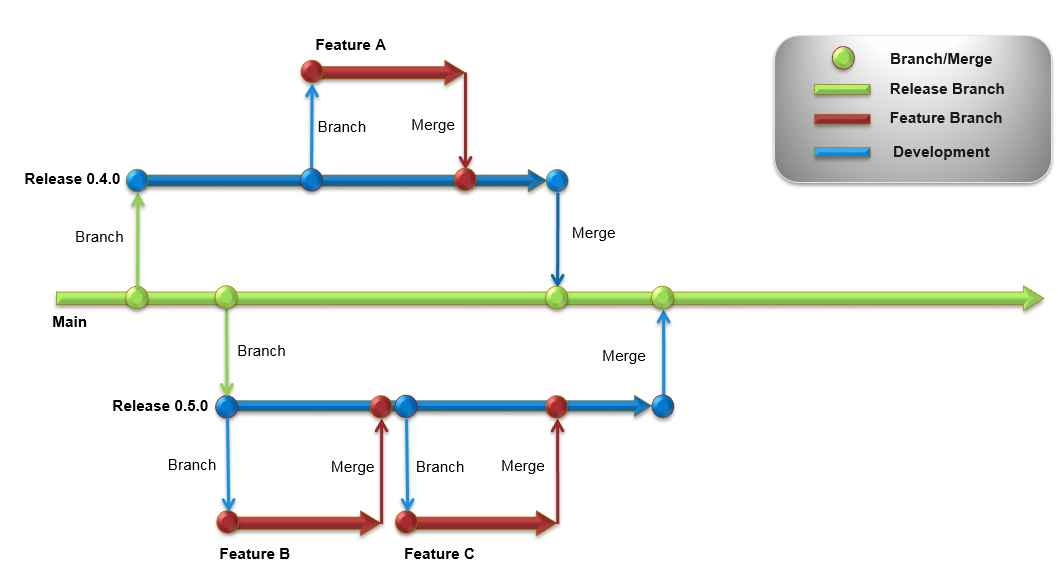
## Branch by Feature

A simple way to explain the “*Branch by Feature*” for different developer teams is in the image below.   
Development branch **Release 0.4.0** is branched out prior next planned **Release 0.5.0**.

In this case both developer teams are working towards their own release schedule and their own features independently from each other.

The **Feature B & C** team would possibly encounter code merging issues upon merge towards the Main-branch, since their release occurs at a much later point, in cases were developers stepped into the same areas of code the merging tool would request these merging issues to be resolved upon this merge.

A feature in these examples can be a single part of the system such as a system module, an integration to/from another system or a system function effecting one or more parts of the system as a whole.



Feature releases reside always in their own development branch and pushed to the central repository (Main). In difference to “Branch by Release” strategy, Branch by Feature is using their development branch as their parent branch.

When features are completed they get merged back into their development branch without interacting immediately with the Main-branch.

Once the development branch has gathered enough features for a release (or pre-determined date for a release), the fork is branched back into its main branch.

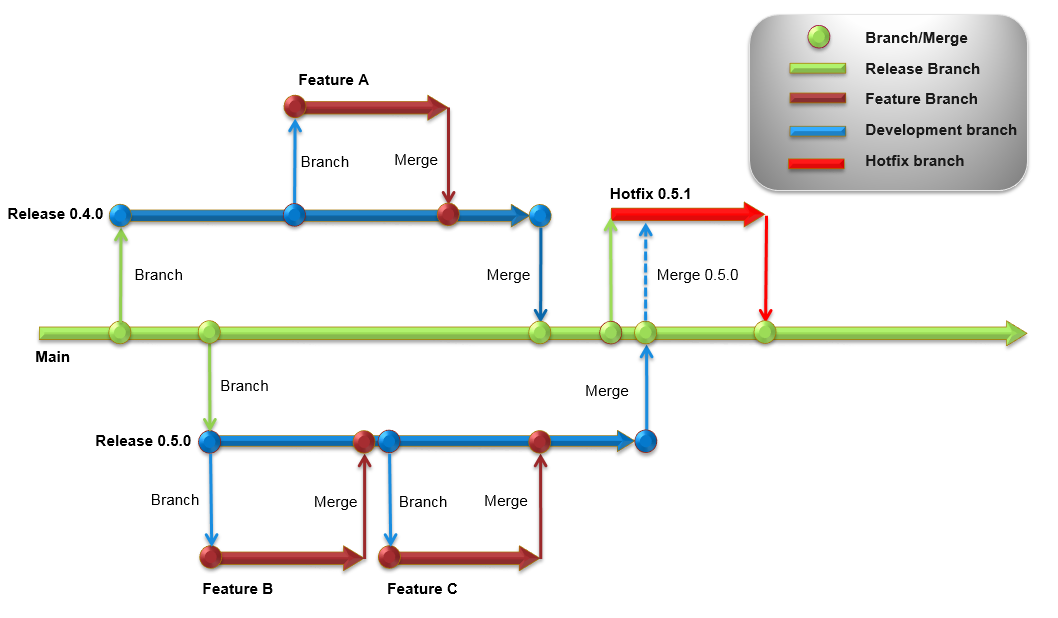
This allows more complex scenarios of multiple teams working with parallelism without interfering with each other. However coordination and communication together with strictly pre-determined release plans are crucial elements in order to gain full control of code merges.

In the case of Skånetrafiken project a Release by Feature strategy would meet the needs of smooth development and multiple teams involving also the maintenance team for bug and hotfix releases.

## Branching for maintenance

All branches aimed for maintenance should be marked with the prefix “Hotfix”. A hotfix branch is the only branch that will allow development by parenting the Main-branch.

A hotfix branch should not be extended to more than 3 – 5 working days. If the time for a hotfix is extended to more than a working week, it should belong to mainstream development and be part of the current running development plan that is following an iteration cycle.



In the example above although the hotfix branch started while the bug was reported in 0.4.0 version of the system, the planning in delivering the fix is ahead of the 0.5.0. Therefore it is the delivery planning that decides the naming of the hotfix branch.

If a planned delivery comes in between the hotfix branch need to take in consideration these incoming changes from the mainstream development (example, merge 0.5.0 from the image above).

## Service Center Case Handling

Service Center handles cases by type and by priority. Depending the severity and priority of the case these can be sorted into 2 main categories.

* Problems caused by development and system delivery.
* Problems caused either by network failure, authorization & configuration issues, network maintenance, OS-patching, etc.

In the case were problems are caused by developers and system deliveries, there’s a warranty time of maximum 4 weeks after a system delivery taken place (in accordance to the customer contract) were the streamline development group is taking care of the discovered bugs and environment issues immediately after a system delivery.

Therefore when programmatic errors that’ve been discovered after the warranty period is over, usually have low severity/impact on the system and would be automatically classified as minor. Bugs based on these criteria should be planned if possible with the mainstream development and be delivered as being part of mainstream development.

In cases when maintenance developers can either not join the mainstream development branch, or for some reason their bugs have high severity and high priority a hotfix branch should be created off the Main branch.

Hotfix branches are the only type of branch that can allow parenting immediately to the Main branch. Hotfix branches should also not extend to more than 5 working days.

Work items should always be registered in the Features backlog and be of type “Bug”. The

# Versioning

## Versioning Control

In order to obtain higher quality of code the following functions should be enforced by team foundation server versioning control.

* Gated Check In to ensure your code is compatible with the development branch.
* Checking code in should be made often.
* Developers should do get latest often.
* Proper comments to identify changes easier.
* Work item association for traceability.
* Code reviews to ensure coding guidelines are followed and code lifespan is prolonged.
* Code branching from the immediate parent branch should be performed at least once a week during an iteration.

## Branching

At the point of where an iteration in a release development branch reaches its end of the lifecycle, it is important merging the code towards the main/production branch and pushed towards delivery. The development branch is by then locked and a new release development branch is born named after the corresponding release version.

Other depended child branches such as feature branches are also to be firstly merged towards their parent release development branch before they’re get a locked status.

Security and monitoring of branching and merging is out outmost important for consistency and quality assurance in coding.

Changesets together with comments and work item association towards requirements/product backlog items and test cases will strengthen the traceability of future deliveries as well as governing the progress in a project.

# ALM Artifacts

## Product Backlog Item

Product backlog items are to represent smaller pieces of requirement entries. They should be only functional requirements over user stories and supported by user experience (UX) or storyboards.

A product backlog should also be backed up with a priority and a level of severity in the list among other PBI.

## Kanban Board

A Kanban board is the working space area for the entire application lifecycle and monitoring of on-going progress that answers to specific iteration resulting to a quality assurance phase and further to a delivery/release in production environment. Thanks to the Kanban board and to the team’s active use upon daily basis enables the project and team management to act upon decision making, leading the developer teams towards a specific direction.

## Work Task

Work tasks are defined as the minimum parts of a technical solution that answer to a backlog item/functional requirement. Work tasks can be divided into Areas, have a priority level, an original estimate and are normally assigned to one of the members in a development team.

## Test Case

A test case documents the pre-conditions and steps to follow in specific order of actions in regards to input/output in a system or part of a system that meet finally the original requirement specification. Test cases can be linked to PBI as child-parent relationship making it easier for testers to check requirements, read user stories and looking up eventually at UX or storyboards provided form the beginning, so that they can answer better to rejected or re-opened bugs/requirements upon a quality assurance testing phase.

# Delivery Procedures

## CRM Delivery Procedures

* CRM solutions are to be always delivered in managed mode towards the following environments
  + **Test**
  + **Quality Assurance/Acceptance Environment**
  + **Production Environment**
* Any of the following items should be added in development environments prior a release and never in any other environments at higher level.
  + Web Resources such as a Javascript, HTML, images
  + Plugins
  + Silverlight controls
  + SSRS Reports

## WCF Delivery Procedures

Prior release WCF services are to be merged from their parented development branches into the main/production branch. Make sure to produce executables in release-mode. A backup of the currently installed files should be placed under the backup folder before proceeding deploying the newer version.

## Database Delivery Procedures

Alteration of different database objects should always be scripted and placed under the release folder structure ready for further deployment and execution.

At no point should changes be made manually at the stabilization, quality assurance or production environments.

## SQL Server Reporting Services Delivery Procedures

SSRS reports should be placed under the release folder structure ready for further deployment on the release date. At no point should SSRS report releases be performed individually without passing stabilization and quality assurance testing first.