<Company> Internal

<ProjectName>

Configuration Management Plan (CM_Plan)

Author(s): <Nombre Autor>

Document Version: 1.1.0

Revision History

Version	Date	Summary of Change	Author
1.0.0	6-Sep-2010	This version is equivalent to X.10 baseline of old version format.	<Autor name>
1.1.0_Draft_A	23-May-2011	Refactoring of CM_Plan with major updates.	<Autor name>
1.1.0_Draft_B	18-Jul-2011	Rework after comments	<Autor name>
1.1.0_Draft_C	25-Jul-2011	Rework after more comments	<Autor name>
1.1.0	26-Jul-2011	Baseline version.	<Autor name>

Approval Page

Following are listed persons which their approval shall be requested for each major change of this plan.
* Approvals are not necessary in case the change is minor.
* CMs approvals are only necessary if the change of this plan is performed by other person than the CMs.

Engineering Manager	Date	Signature
<Person Name>		

UberScrumMaster	Date	Signature
<Person Name>		

Release Manager	Date	Signature
<Person Name>		

Process Manager	Date	Signature
<Person Name>		

CM / CM Backup	Date	Signature
<Person Name> / <Person Name>		

Table of Contents

1 Introduction

1.1 Purpose and Scope

This document covers the Configuration Management Plan for <PROJECT>. The intent of the CM Plan is to control the configuration of the requirements, documents, hardware, software and tools used for this project.
For all tools that are distributed from <Organization name> tools group or corporate <Company>, the project will follow the respective group's Tools Management systems rules.
SCM is the process by which methods and tools are identified to control the software throughout its development and use.

1.2 Purpose of SCM Practices

- Ensure consistency in practicing SCM activities
- Define the authoritative bodies to support the SCM practices
- Maintain product integrity throughout the life cycle
- Inform affected groups and individuals on project status
- Create a verifiable history of current and prior states of work products
- Process improvement

1.3 References / Acronyms / Glossary

Reference	Description	Links/Address
[<PROJECT>_Compass]	<PROJECT> Compass Site	http://...
[CCB_Link]	CCBs link	http://...
[<PROJECT>_RTC]	<PROJECT> RTC link	http://...
[<PROJECT>_Bamboo]	<PROJECT> Bamboo link	http://...

Table 1 - References

Acronym	Description
BO	Business Operation
CCB	Change Control Board
CILOC	Configuration Items Location
CM	Configuration Management
PCM	Project Configuration Management
PSO	Professional Services Organization
SI&T	System Integration and Test
SOP	Standard Operating Procedure
SCM	Software Configuration Management
Tag	a.k.a labels. Identifiers for a Configuration Item
CI	Configuration Item

Table 2 – Acronyms/Glossary

1.4 Configuration Management Tools

Tool / Process	Purpose	Tool Controls
Compass	Project status reporting and document management and control for versions of documents.	Electronic revision control. Check-in/Check-out.
CVS	Configuration management tools used by Software, Hardware, and Quality Engineering groups for control of source code and build environments.	Electronic revision control. Check-in/Check-out. Source code branching and merging capabilities for parallel development.
Clear Quest	Computer-based requirements and change management tool; used by Product Management, Development engineering, and SI&T.	Change requests.
Rational Team Concert	Integrated workflow management for agile teams	Assignment of tasks, progress tracking, status dashboard
IBM Architect	Design and architecture diagrams (UML et al)	N/A NOTE: Recommended tool for architectural diagrams
Bamboo	Continuous integration build server.	Builds are automatically triggered based on changes to the repositories. Manual build generation.

Table 3 – Configuration Management Tool

2 Configuration Management Roles

2.1 Project Configuration Managers

Configuration management activities will be coordinated inside <PROJECT> project by the Global Project Configuration Manager (GPCM) role which will be assigned to one person. Additionally, a backup GPCM will be designated inside the team as well.
The Global PCM will be responsible for activities like tracking the main and release branches, determining when branches are created, what development activities belong on what branches, etc. In customer and standard package modules, PSO will have ownership but the GPCM will support and help them to implement the CM practices defined in this plan.
Also development teams, as for example scrum teams, may have their own Project Configuration Manager (TPCM) reference to conduct the CM activities pertaining to the team and assist the GPCM with the broader project-level and scoped to his corresponding sprint branch.
Configuration management activities, process, procedures and policies must be followed by all team members. It is the responsibilities of each person to follow and apply the proper CM process, according to its assigned role/roles.
Following table shows persons that will be taking the role as Global Configuration Managers:

Role	Primary	Backup
Global PCM	<CM Eng. Name>	<CM Backup Eng. Name>

Table 3 – Global Configuration Managers

2.2 Configuration Management responsibilities

Role	Responsibilities
GPCM	Possess overall responsibility for all configuration items Responsibility for the creation of all branches and administering their policies Responsible for the application of labels in main and release branches Ensure product integrity and traceability of the configuration items for the whole project Coordinate CM activities inside the project. Ensure the correct execution of the CM schema. Assist in merge activities to the main and release branches. Building activities at main branch and release levels. Participate in audits. Analyze all CM related findings.
TPCM	Assist in creation of tags and branches. Assist in merge activities for user stories to the main branch Building activities at team-specific branches Ensure product integrity and traceability of the configuration items owned by the team Participate in audits. Analyze all CM related findings.
Team	Help to resolve conflicts during the merge activities. Ensure that quality criteria for the deliverables to the main branch are met Follow all associated processes, policies and practices defined for their assigned roles.

Table 4 – CM Responsibilities

3 Change Management

3.1 Scope

Change management is a process that occurs after documentation, source code or product hardware baseline is identified and approved. Changes include internal changes to the original documented approach due to simulation or test results or external requests for changes to features or functions.

3.2 In Core Releases and Customer modules

3.2.1 Technical CCB (Change Control Board)

The T_CCB is a committee that ensures every change is properly considered by all parties and is authorized before its implementation. The T_CCB is responsible to approve, monitor and control change requests to establish baselines of configuration items.
The items to be reviewed by the T_CCB are those changes brought by the Issue Coordinator, the T_CCB chair or other T_CCB members.
The scope of work will be to approve/reject changes needed in plans, documents and code. Decisions shall be made about actions to be taken based on the result of product quality assurance activities and product health after each test cycle.
CCBs minutes will be kept in **[CCB_Link]**.

3.2.1.1 Members

The following table shows the team members that attend Technical CCB meetings. The ones specified with * are mandatory, other ones are optional members and will be invited if necessary.

T_CCB Role	Name
Engineering Manager - CCB Chair	<Eng. Name> (backup <Eng. Name>) *
Release Manager - Issue Coordinator	<Eng. Name> (backup <Eng. Name>) *
Engineering Manager	<Eng. Name> (backup <Eng. Name>)
Ubber Scrum Team	<Eng. Name> (backup <Eng. Name>)
PSO representative	<Eng. Name> (backup <Eng. Name>)
Engineering Director	<Eng. Name> (backup <Eng. Name>)
GPCM	<Eng. Name> (backup <Eng. Name>)*

Table 6 – Technical CCB Members

3.2.1.2 Frequency

CCB Meeting	Frequency
<PROJECT> CCB	Meeting held weekly or on a demand-base.

Table 7 – T_CCB Meetings Frequency

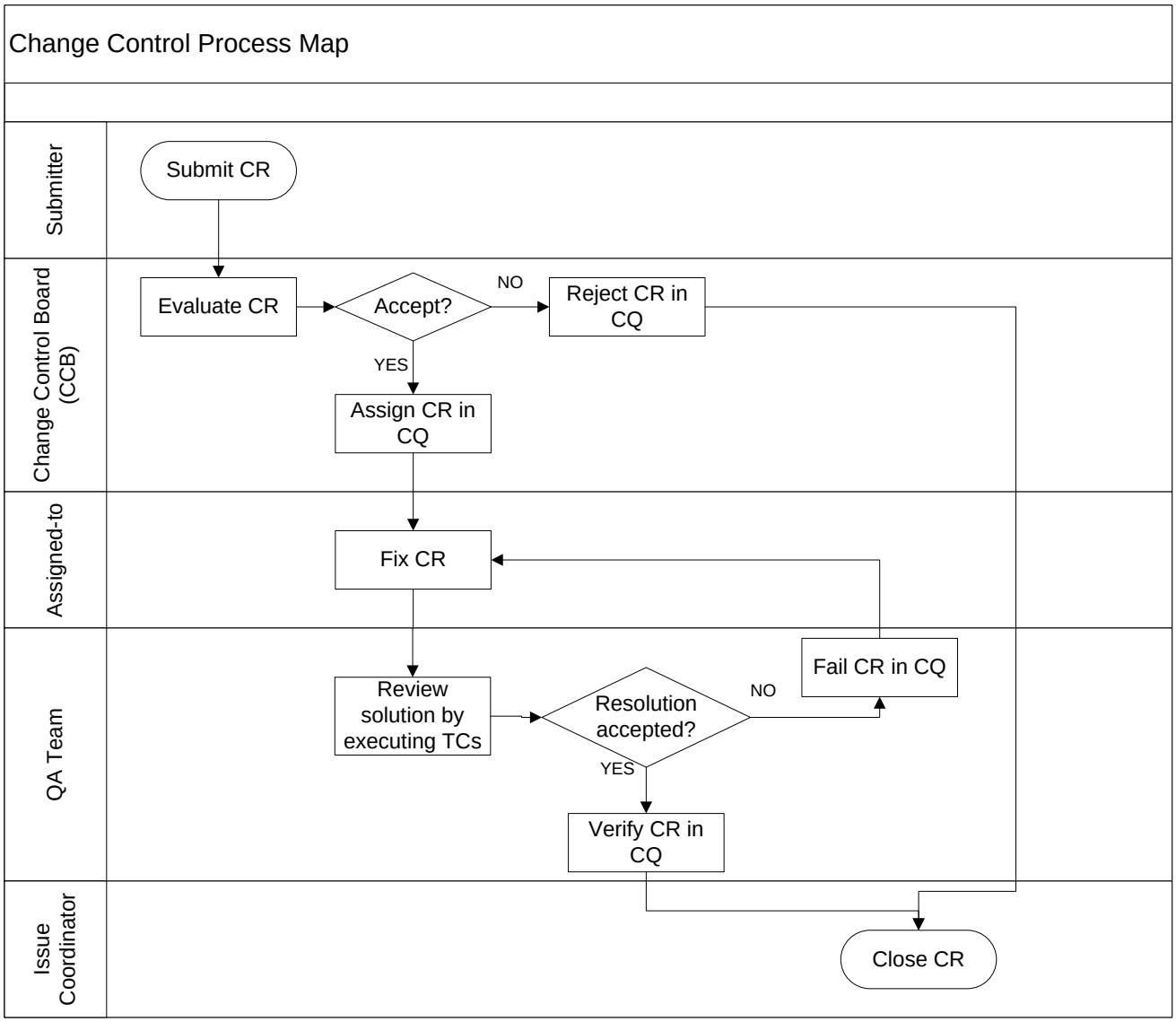
3.2.1.3 Change Management Tool

A ClearQuest instance will be used to manage Changes and Enhancement for core releases and customer modules:

Field	Value
URL	http://...
Schema Repository	'cq#432'
Database	'cqqlewn'

Table 8 – ClearQuest Instance information

3.2.1.4 Change Control Process Map



3.2.1.5 Considerations:

- The IC shall determine whether the CR is duplicated of another CR, in which case it shall be referred to the parent CR.
- The IC can only reject a newly submitted CR if this issue was previously discussed in a CCB meeting. A CR will be rejected after the CCB members agree on it.
  If the CR is rejected, the IC shall give a reason for the rejection.
- If a CR is a minor/cosmetic issue or has a straight forward solution that does not need to be discussed during the CCB meeting, the IC can assign the CR previous to the meeting taking into account the following conditions:
  - The IC will send a list of the CRs that were assigned previously to the meeting for CCB members review (the IC must communicate all assignments performed and not discussed in a CCB meeting)

- During the CCB meeting, team members agree on the previously assigned CRs, otherwise the not agreed CRs are discussed during this meeting

3.3 In Scrum teams

In scrum teams the track of changes is performed using RTC tool following agile process. For details please refer to **[<PROJECT>_RTC]**.

4 Configuration Identification

Compass Site will be used to keep and share main assets of different releases and teams, refer to **[<PROJECT>_Compass]**.
General plans, documentation and items that apply for all releases will be kept under the folder <PROJECT>\<PROJECT> Process & Project Plans.
To differentiate per Teams, also it can be used as <PROJECT>\<PROJECT> Process & Project Plans\<Team>.
For each Release, a release folder will be created as <PROJECT>\<RelID>.
When it is necessary, teams that are involved in a specific Release may keep their main assets into Release folders <PROJECT>\<RelID> and follow their own structure.
To differentiate per Teams/Components, also it can be used as <PROJECT>\<RelID>\<Team/Component>.
Customer Releases folders <PROJECT>\<RelID>\<Customer> will be created to keep main deliverables for Customer Releases.
It is recommended that every team creates a Configuration Item Location document (CILOC) definition to specify where main assets of that team will be located under Release folders.
If a team decides defining a CILOC document, it can be located under the <PROJECT>\<PROJECT> Process & Project Plans\<Team/Component>.
There are other CIs such the ones belonging to StdPkg and User Documentation that have specific location: <PROJECT>\<PROJECT> Standard Packages and <PROJECT> (<PROJECT> Tech Pubs Page link).

5 <PROJECT> Teams

The scope of this document covers source code managed by the following teams:

- **Scrum Teams:** they are in charge of developing new features and functionalities through stories following the agile process. They will commit their code in sprint or feature branches so then be merged to the integration branch at the end of the sprint. Scrum teams also can be assigned to do extra work as bug fixing for core or customers modules, in those cases they will commit in the corresponding branch that is receiving those fixes.
- **Release Management Team:** RMT is in charge of performing all necessary testing to turn the core products Release state that will be defined according the release to be used then by customer components. Bug fixing of what the RMT finds will be committed on branch where core release builds are being done.
- **PSO Teams:** these teams are responsible for the delivery to end customers of the available product release. They will add specific customizations and scripts in the corresponding customer module that will be based on a specific core, das and standard package versions.
- **Rapid Reaction Team:** this team is responsible for addressing customer issues requiring urgent resolution in the form of hot fixes (i.e. that cannot wait to be incorporated and delivered with the next scheduled release) and these fixes shall comply with the DONE criteria. RRT team also can be assigned to do extra work as development of stories and bug fixing for core or customers modules, in those cases they will commit in the corresponding branches that receive those stories or fixes.
- **Components Team:** the goal of this team is to find out and identify different applications and modules that can be integrated to EGDE that add value to the product. Generally they work in dummy repositories until it is decided and approved to be implemented into production repositories.
- **Product Documentation Team:** this team is responsible of creating and maintaining the product documentation that will be delivered to the customer. Part of that documentation is integrated as part of the <PROJECT> product in the source code repositories and other part is maintained in Compass. Example of assets they manage: Admin guide, User guide, online Helper in <PROJECT> product, etc.

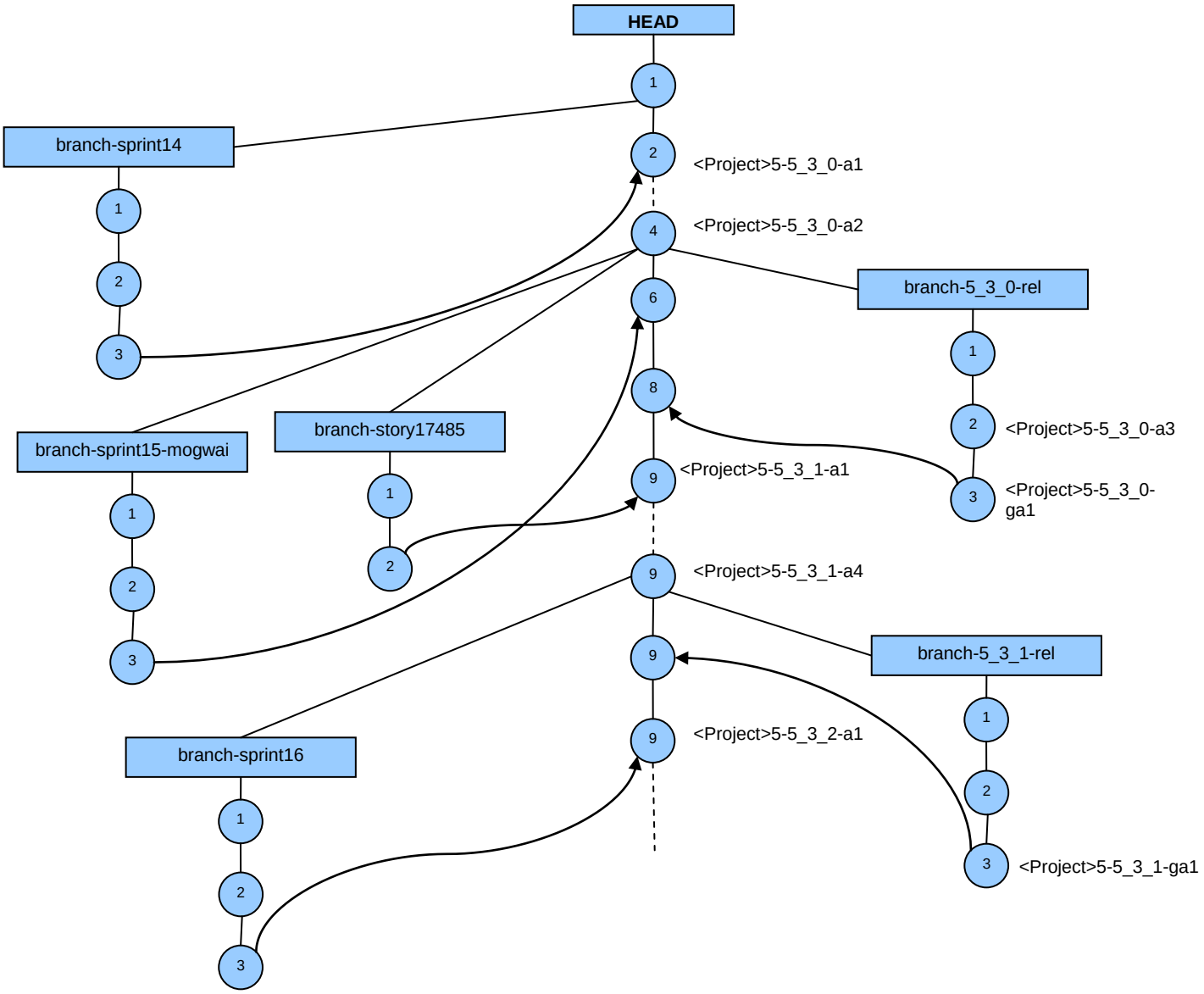
For detailed information about branching schema refer to next section.

6 Source Code Configuration Management

In this section different items of source management are described. It covers some aspects about branching schema, tagging, merging strategy and expected quality levels for the whole product.

6.1 Core modules

Graph below shows branching schema that will be followed in Core modules.



6.1.1 Branch definition:

Types of branches that will be utilized are defined as follow:

- Integration branch: the HEAD is defined as the main integration branch where all features released will be held. This branch is created by default by CVS when a file is added to source control.

- Development branches: they are those branches where development of new features/functionalities is coded. Development branches that can be utilized are:

  - Sprint branches: this type of branches shall be created by the CM and they are utilized by scrum teams. Here development of stories will be committed until they are merged to the HEAD branch. It is recommended that they are created one per sprint * and they can be created also per team and/or many scrum teams can work together in the same sprint branch according the circumstances. In general, scrum teams will work in the same sprint branch when features of those scrum teams are going to belong to the same release, anyway that approach is not mandatory and will be defined according the needs.
  Name format of these branches is: branch-sprint<#>[_team] where team value is optional, for example: branch-sprint15, branch-sprint16-<teamname>, etc.
  * It may be decided that a sprint branch has related more than one sprint, it might be so if for example there are many stories in Not Done state and they can be completed in next sprint so it is decided to keep the development of that next sprint in the prior sprint branch until they are more stable to be merged to the integration branch. Other example can be that stories belonging of that sprint branch are thought to be included in a release that still is not planned to be formal built for the RMT and there are prior releases that are ongoing and using actively the integration branch. So supposing that one of these cases occurs, for example a branch-sprint7 can be used for sprint7 and sprint8.

  - Feature branches: although most of features should be developed in a sprint, feature branches might be used in case a specific feature is complex or is a high risk feature. They can be used to not break stability of some branch or to try it individually until it is as stable to be merged back to the corresponding branch. These branches can be created by the CM or by the scrum team.
  Name format of these branches is: branch-<feature> where feature value is a word that helps to understand about the feature is. Some examples: branch-jmsless, branch-OSBG-merge, branch-story15423, etc.

- Release branches: this type of branches shall be created by the CM and they are created to state here the code versions of specific release and they will be branched off from the HEAD. The Release branch name will be **branch-<version>-rel**, for example: branch-5_2_3-rel, branch-5_3_1-rel, etc.
  * If a minor release is the continuation of a prior release, it may be decided that the new release version is continued in the same branch of the predecessor release, for example: release 5.2.1 was committed in the branch-5_2_1-rel, then there is a release 5.2.2 that is the continuation of 5.2.1 with minors fixes, so 5.2.2 is committed in the same branch-5_2_1-rel, and so on.

Having defined types of branches, the main idea of the branching schema is to keep the isolated unstable code in development branches against the stable code potential to be released. As shown in the graph above, the latest version of the product will be the owner of the HEAD to perform its bug fixing until a new parallel version is ongoing, older version will be isolated in a release branch so its bug fixing continues there. When it is decided, bug fixing of older releases should be propagated to newer release versions.

6.1.2 Tag definition:

- Root tags: every time any branch is created, a tag formatted Root_<branch_name> shall be applied on the source version from where the branch is created; if the branch is created using Eclipse tool and is not based on an existing tag, this tag is applied automatically; if not, apply it manually. Examples: Root_branch-5_2_1-rel, Root_branch-5_3_1-rel, etc.

- Merge from tags: they are suggested to be applied every time a merge is performed in the source versions. The format is merge-from-<source>-<yyyymmdd>-<nn>. Examples: merge-from-5_2_1-rel-20101205-01, merge-from-story12748-20110222-01, etc. For details, please refer to merge strategies section.

- Pre-Merge tags: this type of tags is optional. They are suggested to be used when a very complex merge is going to be performed. The idea is to apply this tag on the branch/version that is going to receive the changes before the merge is performed, in that way, we will have marked the stable versions just in case the code gets harm after the merge and so we will be able to identify the prior stable versions. The name format is pmerge-from-<source>-<yyyymmdd>-<nn> and it has to be associated to a Merge from tag corresponded to the same merge. Examples: pmerge-from-5_2_1-rel-20101205-01 (associated to merge-from-5_2_1-rel-20101205-01), pmerge-from-story12748-20110222-01 (associated to merge-from-story12748-20110222-01), etc. For details, please refer to merge strategies section.

- Build tags: they will be applied every time a formal build is created. These tags are used in order to have versions identified that are used to create an specific build and they shall be applied automatically when the corresponding Bamboo Blessed plan is run by the authorized CM.
  The format of this type of tag depends on the CVS customer/component repository:
  - in bezoar module: <Project><major_version>-<version>-<buildid>, for example: <Project>5-5_3_1-a1, <Project>5-5_3_2-ga1.
  - in customer/component modules: <Project><major_version>[-customer/component]-<version>-<buildid>, for example: <Project>5-Telefonica-5_3_2-ga22, <Project>5-Telecom-5_3_1-a3, <Project>5-das-5_3_1-ga2.
  - in <Project>5_standard_package module: release-<major_version>_<buidid>, for example: release-1_78, release-1_81, release-2_8.

6.1.3 Auxiliary CM files:

There are 2 auxiliary files that might be used in <PROJECT> CVS repositories for CM purpose, they are:

- MergeNotes.txt: in this file are registered the history of merges performed in some specific branches with some necessary information.

  Examples of that information:

  2011-04-18 (<dev.name>) Merge from branch-5_3_0-rel to HEAD
  End tag on branch was merge-from-5_3_0-rel-20110418-01
  Start tag on branch was merge-from-5_3_0-rel-20110405-01

  Bug fixes included in this merge:
  cqdsl00048072 -- Added selected device to deep links from CSUI.
  cqdsl00046770 - <custmertool> 192458 - [Lot 1] GP / ANY / Some "INSERT INTO Customer" failures Infrastructure to optionally suppress <PROJECT>.2001 alarms was implemented.

  2011-05-04 (<dev.name>) Merge from branch-sprint14 to HEAD
  End tag on branch was merge-from-sprint14-2011-0504-01
  Start tag on branch was Root_branch-sprint14

  This merge represents the termination of sprint14.
  Story 17161 <Feature description 1>
  Story 18638 <Feature description 2>
  Story 15235 <Feature description 3>
  Story 17931 <Feature description 4>

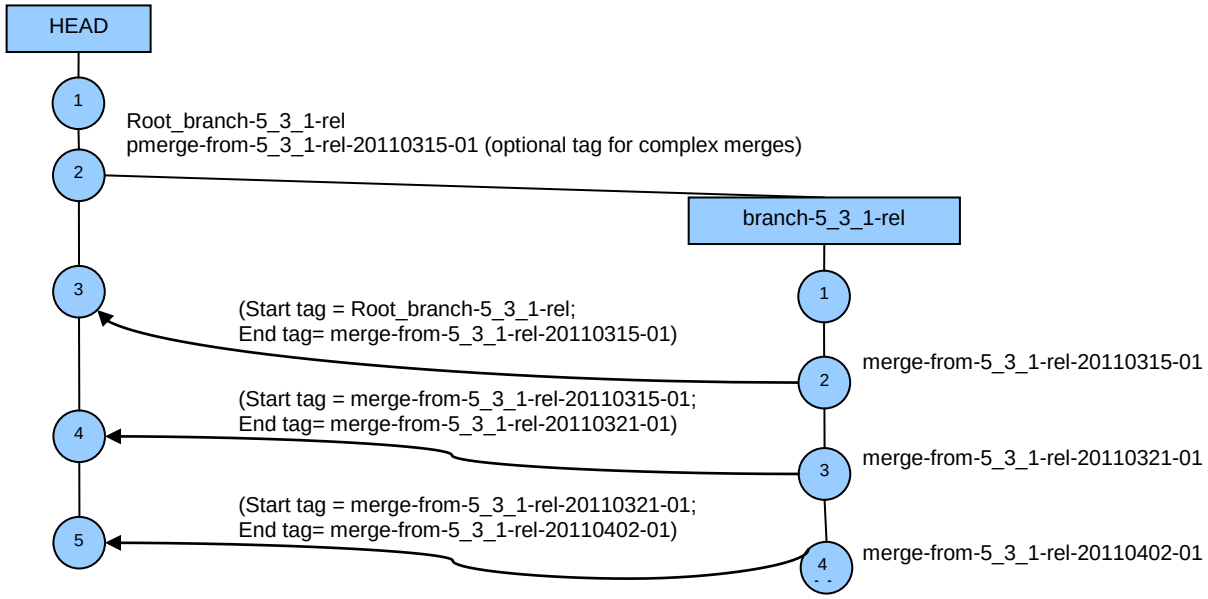
- ReleaseNotes.txt: this file is used in some branches to have a track of list of changes introduced in a branch. Developers shall add a note on the top in this file about the change they are committing. The mergedfrom field only will be used by CM that performs the merge activity.

  Example of that information:

  date: 2011/05/10
  author: <dev.name>
  trackingnumber: story 17161
  mergedfrom: <Project>5-5_3_1-ga2
  modules: Core
  status: Done
  title: <Feature description 2>
  comment: I Have finished the Unit Testing; Also the call to deleteCpe function was moved: after the profile has been set.

6.1.4 Merge strategy:

To perform merges in cvs modules, you have to specify the End tag (version to be merged) and the Start tag (common version).
Before performing a merge, it is recommended to apply merge-from tag in source versions especially in those branches which is likely that more than one merge can exist. Merge-from tags are used to identify versions that where merged so in case another merge is needed in that same branch, we are able to use the last merge-from tag applied in the prior merge as the Common base version (start tag) and so avoid dealing with the same conflicts that we resolved in the prior merge.
Also optionally, you can apply a Pre-merge tag in case you have to deal with a very complex merge so you can have identified the stable versions before the merge is performed.



For example in the graph above, we need to merge the branch-5_3_1-rel to the HEAD to propagate some bug fixes to the integration branch.
The first merge is performed at 15-Mar-2011 and Start tag used is the Root_branch-5_3_1-rel and the End tag is the merge-from-5_3_1-rel-20110315-01 that was just applied before the merge. For this merge also it was decided to apply the Pre-merge tag pmerge-from-5_3_1-rel-20110315-01 before perform the merge since it is a complex merge just in case we need to identify the prior stable versions.
Then we need to perform another merge from the same branch because other fixes were committed, so the Start tag used is the merge-from-5_3_1-rel-20110315-01 (the one utilized as end tag during prior merge), and the End tag is the just applied merge-from-5_3_1-rel-20110321-01. In this second merge we can see how the start tag is specified by the merge-from tag used in the prior merge, that way we'll be only analyzing the gap introduced and avoiding dealing with conflicts that were resolved in the prior merge.
In third merge the same approach as the second is followed, but start tag is merge-from-5_3_1-rel-20110321-01 and End tag is merge-from-5_3_1-rel-20110402-01.

To register the merge history and Start / End tags utilized, we can use the MergeNotes.txt file which is located in the root of the project; this file will have its own version evolution in those branches that receive the changes from a merge and it is necessary to track this information, specially the HEAD branch. As optional information, list of changes merged can be listed. (See auxiliary files for details).

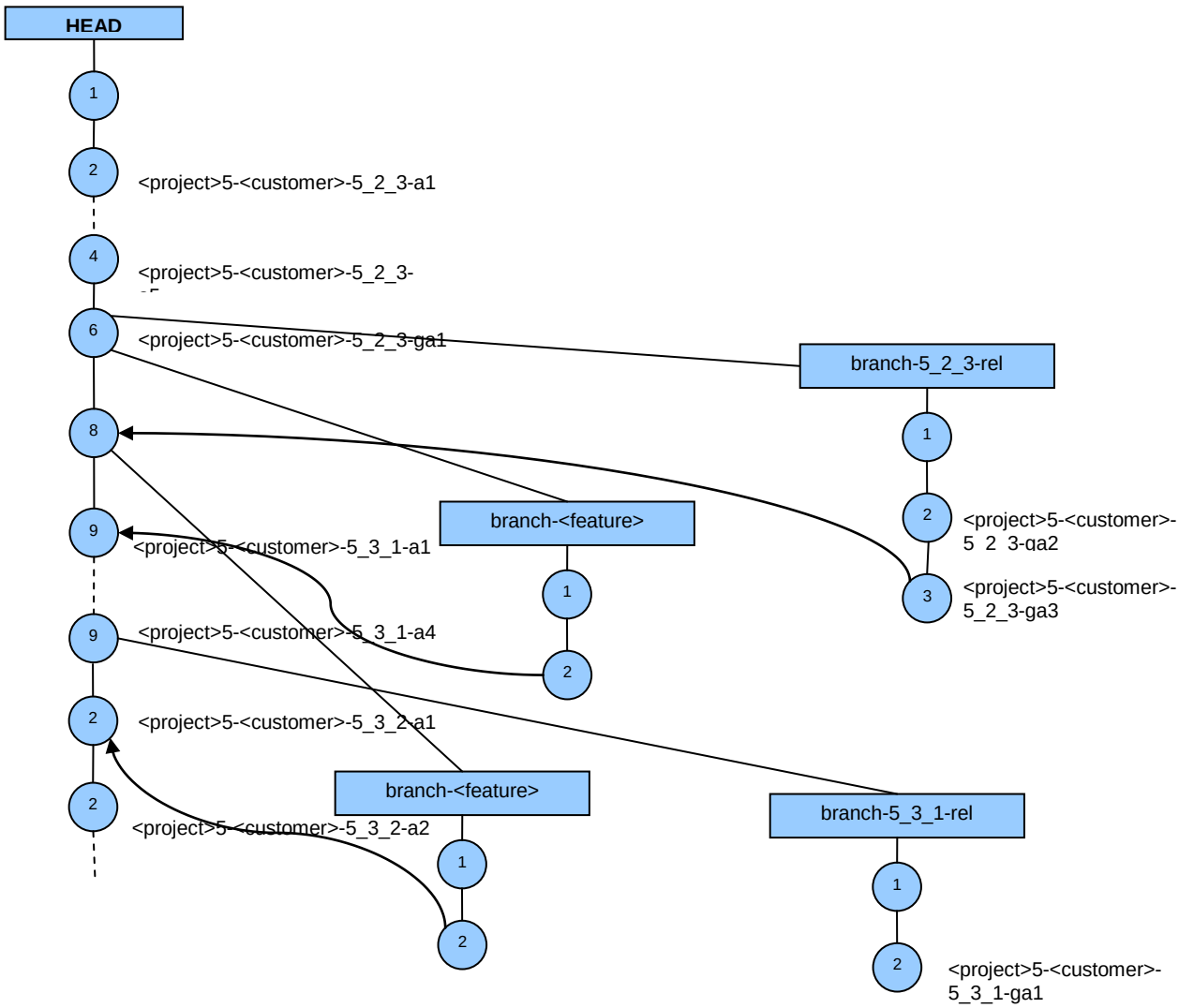
Also as part of merge activity, if applies, we can update the ReleaseNotes.txt file with list of changes that are being propagated to the branch so we can have the track of list of changes just introduced during the merge.

Merges from sprint branches to the HEAD or feature branches to the HEAD can be performed by the CM or by an expertise developer tracked by the CM.
A scrum team can decide to isolate some stories in different feature branches during a sprint, so then the same scrum team can perform the corresponding merge from those features branches to the corresponding sprint branch or advice to the CM to merge it to the HEAD directly depends on what merge strategy is decided by the CM.
Merges from release branches to other release branches or to the HEAD to propagate fixes shall be performed by the CM.

To see details of MergeNotes.txt and ReleaseNotes.txt files refer to Auxiliary CM files for details.

6.2 Customer modules

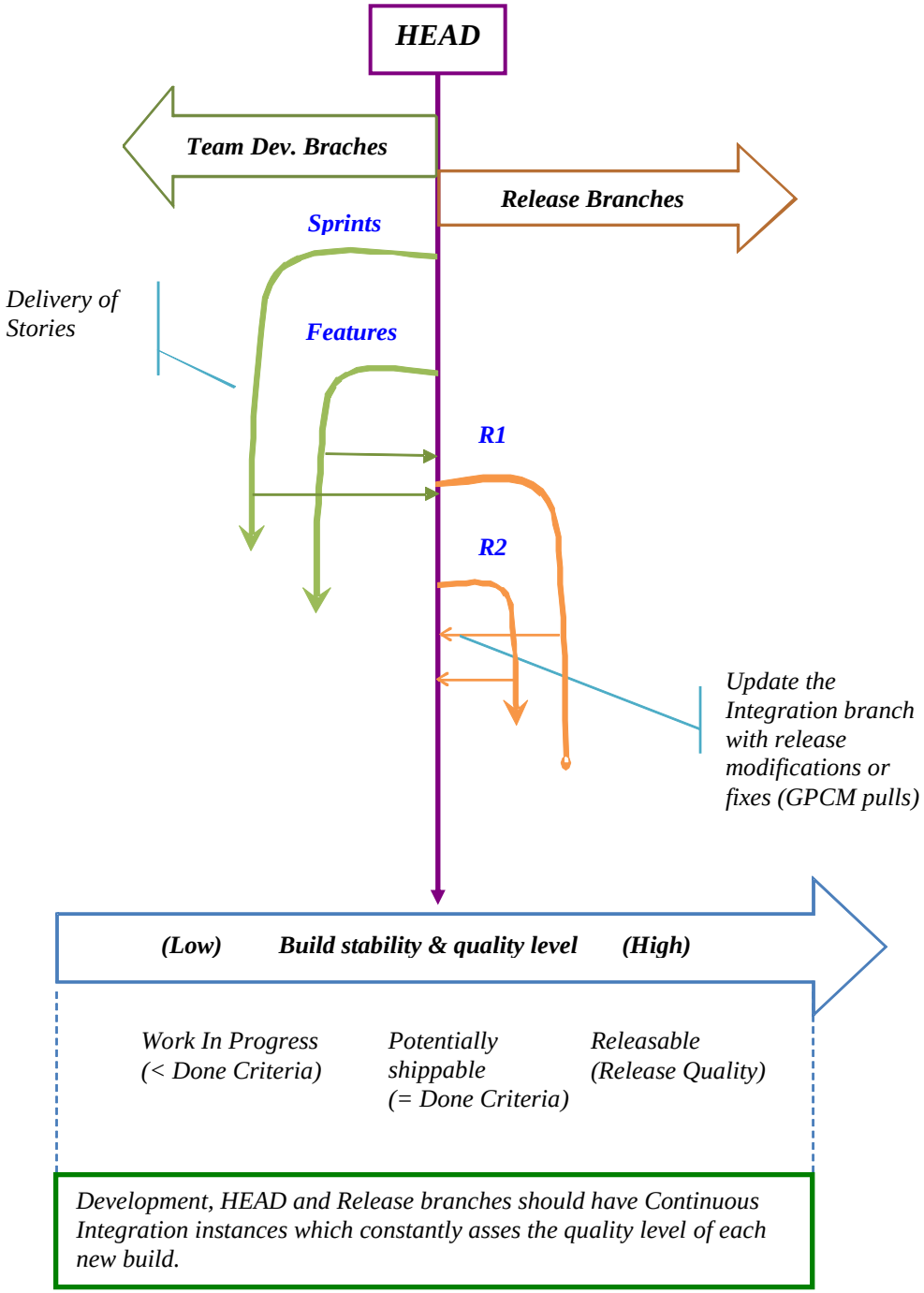
For Customer specific changes, there will be a module per customer where they will evolve by its own. In these modules all previous CM concepts apply except that development of new features here will be only using feature branches and not sprint branches since the owner of these type modules is the corresponding assigned PSO team and they don't follow a scrum process. Also for first version of the customer might happen that all initial development is done in the HEAD until a new parallel version comes in and a new release branch is needed.
Branching schema for customer modules is shown in the following graph:



A peculiarity of customer modules is that they need to be based on a Core and Standard Package version. During the Customer Blessed formal builds the CM will setups those values so Core and Standard Package versions utilized are identified and can be reproducibles.

6.3 Quality levels and Done Criteria

The following diagram depicts the expected quality levels for the whole product.



As shown in the figure above, Development branches (sprint and feature) will be low quality until they are ready to be merged according the Done criteria.

Integration branch (HEAD) branch should fulfill at least the Done Criteria completed since that is the place where potential release versions will be taken to be carried out to the real Release quality level by the corresponding QA team. An exception of that can be that sometimes it may happen that some stories of the development branch that is being merged to the HEAD are in Not Done state, that may be decided because the cost of merging partially a branch with only Done Stories and the movement of the Not Done stories to the next sprint branch is too high, so an approach to follow might be to merge all stories (Done and Not Done) to the integration branch and then they will be completed in the next sprint or Release branch as bug fixing.
Done Criteria will be maintained in RTC, you can find it **[<PROJECT>_RTC]**.

7 Build Management

Types of Build defined are:

- *Continuous Integration Builds:* they are executed on those branches that demands tracking the code is not broken during the development. They are useful to identify as early as possible code errors to be fixed before going into a formal build. They are setup to not apply any tag and to be run automatically for every commit running unit and system tests to generate the corresponding reports that have the results. In the corresponding Bamboo CI instance, useful information can be found like summary of unit and system tests, the list of changed files, the author and the date of the commit, etc. Since they are considered as development builds and their information is not necessary to have any persistence to be reproducible, every certain period of time that information is set to be automatically deleted from Bamboo.
- *Blessed Builds:* they are used to generate formal builds. They don't run unit and system tests and are set to apply formal build tags. This type of builds shall be run only by the authorized configuration management. In the corresponding Bamboo Blessed instance, useful information can be found like list of files versions that changed, author of changes, link to download the build artifacts, Bamboo logs, etc. These builds are persistent and they can be reproducible.
- Informal local builds: these types of builds are created in a development environment and they are created mounting necessary CVS projects on the desired version in the development workspace and running deploy or dist ant targets. Corresponding release.properties and build properties files shall be set according the needs. These will generate a deployable or zip file component. This process will not apply any tag and won't generate persistence builds since they are informal. They are useful for example for developers to check their code before committing or for scrum testers to generate a deployable component to test its functionality. An example could be a developer codes a feature and, before committing, he generates a deployable component to check it works as expected, after committing the code and letting the scrum tester know that the code is ready in the repository; the tester refreshes its project/s and generates other deployable of the component in his workspace to perform the exploratory test.

Having defined the types of builds we'll see how they will be implemented in different branches in all modules:

- In development branches:
  - Continuous integration builds management shall be implemented specially in sprint branches. Feature branches might use continuous integration builds but it's not mandatory, that will be analyzed if it needs a track of its development depending on the situation.
  - Blessed builds here are not mandatory, although they are recommended to be used for exploratory tests and/or to generate builds artifacts for sprint reviews.
  - Informal local builds here can be used, instead of Blessed (or in conjunction), to generate a deployable component for exploratory tests.
- In Release branches: Continuous integration and Blessed builds shall be implemented here since this type of branches will contain the code that will be used to generate formal builds potential to be released.
- In HEAD branch: This branch will have always a Continuous Integration build since it is the integration branch and all product versions will end up here. Also it might temporary have an associated Blessed build in case a version is being developed here until that is moved to a release branch (see section 6: source code management).

* Informal local builds, from the developers' point of view, can be used in any branch to check their code.

The tool that will be used to implement and manage that defined build approach is Bamboo and it will be configured to send emails to different groups subscribers reporting the status of each Build. Information and details about each Build can be found in **[<PROJECT>_Bamboo]**.

8 Release Management

The Release build will be done in the corresponding release branch, starting from the release candidate of choice and adding the corrections of defects found along the system test cycles. For each cycle, the build shall be properly identified and, at the end of the cycle, evaluation of the product health against the defined release criteria will be conducted.
Release builds will be made available in compass and e-mail notification will be sent to the project team distribution list. Refer to section 4 to see the assets location definition.

9 Backup and Disaster Recovery Information

Project documents are backed up according to Compass policies.
<Code> server has a full backup every Monday and Thursday to a tape backup system on the corp net where <ServerAdminsitrator> has access.
Source Code:
cvs backups
The cvs repository is targz'd placed
in /mnt/san/cvsbackups/cvsbackup-date.tar.gd, and rotated Monday-Friday by
/home/cvs/bin/cvsbackup.pl. The most recent backup is hard linked to
/mnt/san/cvsbackups/cvsbackup.tar.gz. The current daily dump is transferred to tape with Retrospect
Retrospect
cvs backups are pushed from cron on cvs.ks.netopia.com as user cvs to
/Users/steven/sourcecode_backup/cvs
Retrospect Schedule
Each backup happens the day after the labeled day. For example Monday backups run on Tuesday morning, Friday backups happen on Saturday morning. Retrospect backs up the following directories Tuesday-Saturday.