Xaraya
Request for Comments: 0028
Category: Historic

M.R. van der Boom
J Schrage
Xaraya Development Group
November 2002

Xaraya Development process: repository model and tool usage

Status of this Memo

This memo describes a historic protocol for the Xaraya community. It does not specify an Xaraya standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This RFC describes the development process used by the Xaraya development Group, both in general process terms as in explicit tool usage terms.

Implementation of policy

With hindsight, since the first revision of this document, the described policy has not been completely implemented so far. My personal opinion is that this is due to the following factors:

- learning curve for bitkeeper
- code is still in flux
- eagerness for producing code

The policy described in (part a) of this document is still the desired policy and I will be working on that process to asymptotically reach the process as described in this document.

However, the most important parts of the process are in place such as the scenario analogy. With the bitkeeper suite of tools, most of this process is not visible but automatically arranged for.

NOTE: Over time we have switched to monotone and this document should be rewritten in a new RFC, this RFC has now been marked as historic.

Table of Contents

1. Repository model

Repository supported scenario based development

The first part of this document describes the policy for the Xaraya source repository. It is written with no particular repository tool in mind. (At the time of writing we use CVS) We will refer to the tool used as SCM (Software Configuration Manager for lack of a better term) to keep this part of the document valid when we decide to switch to another tool. We are evaluating subversion (http://subversion.tigris.org[1]) , bitkeeper (http://www.bitkeeper.com[2]) as replacements for CVS to give us more efficient process support.

To keep the policy readable, the terms used are the ones which are common to CVS. Other SCM tools might use other words to describe the same principles. The actual implementation of this policy with the tool of our choice in in part B of this document.

The document assumes you know how to use a SCM. The text focuses on the rules to guarantee a pristine repository supporting our development team instead of obstructing it. If the SCM cannot support our process, we'll have to come up with some kind of self-discipline guidelines to implement our process. If possible and reasonable we will let the SCM do some sanity checks to keep developers on the right track. Please read these guidelines and stick to them. Each section is accompanied by a section "Possible support/enforcement" which lists a possibility to be able to support the guideline. It does not mean that these measure are in fact in place, but they might be some day!

1.1 Formats

The file format used in the Xaraya repository tree is Unix, i.e. lines end with LF. If you are not using a *nix system please turn off any auto-conversion-to-proprietary-format in your editor, as this will mess up things for other developers and will lead to php execution errors.

Take special care in naming files. Although a *nix file-system is generally case-sensitive and won't complain if you commit a FILE and a file and a FiLe. Windows and Mac systems will not be able to distinguish these files and will not be able to use the repository.

Xaraya uses the PEAR coding standard (http://pear.php.net[3]). Please read these guidelines and follow them when you code. This includes using phpdoc tags for documenting your code (http://www.phpdoc.de[4])

Possible support by SCM:

- check for CR/LF Action: warn/cancel commit;
- signal on pre-commit that same name (case-insensitive) exists, Action: warn/cancel commit
- lint-like tool: Action: warn only
- basic check for documentation tags Action: warn only

1.2 Development process

The development in Xaraya is a team effort. We all agree that the quality of our code needs to be as high as possible. To support this goal we have defined a development process which will be supported by the SCM tool.

Before we describe the process let's define the terms we use to describe it.

[1] http://subversion.tigris.org
[2] http://www.bitkeeper.com
[3] http://pear.php.net
[4] http://www.phpdoc.de

1.2.1 Trees, trunks, scenarios, branches, tags and others

Tree:

The tree is the total of all elements, typically files, containing the sources of the project, which we want to manage. The files are not limited to source code; it is even recommended that all information which is more or less part of the project will be placed under control of the SCM. Think of documents, static web-pages, test-scripts, sql-scripts etcetera.

Trunk:

The trunk represents the main line of development and always contains the latest approved changes in development.

Branch:

A branch is a (temporary) (partial) copy of the tree which is created with a specific purpose. Changes made in a branch do not appear on the trunk unless they are merged back in.

Scenario:

A scenario is a specific development task. A scenario always lives on a branch and is created by the repository managers in the repository.

Tags:

Tags are created in the tree to represent a specific state of the tree which might be needed in the future. For example, you might tag the tree before you commit a very complex change.

Releases:

Releases are basically the same as tags, with the exception that they are permanent, read-only and occur always on a release branch. They represent the known state of the tree and contains the exact contents for a certain release of the software.

1.2.2 Development process description

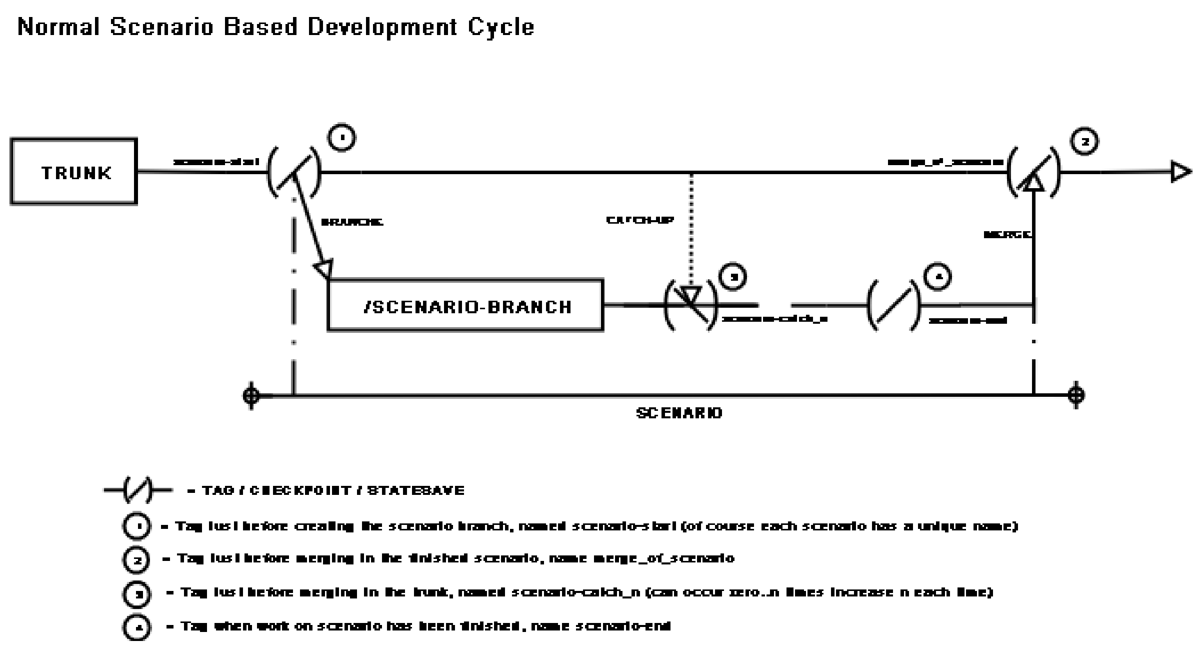
We use a development process we call "scenario based development". In short this means that all development takes place in scenarios (which always live on a branch, remember?). No development takes place on the trunk. I repeat, no development takes place on the trunk ;-)

The process consists of two main activities:

1. Normal scenario based development cycle
2. Release scenario

1.2.2.1 Normal scenario based development cycle

During normal development the following process is used:


Normal Scenario Based Development Cycle
TRUNK
BRANCHE
CATCH-UP
MERGE
/SCENARIO-BRANCH
SCENARIO
- TAG / CHECKPOINT / STATESAVE
1 - Tag just before creating the scenario branch, named scenario-start (of course each scenario has a unique name)
2 - Tag just before merging in the finished scenario, name merge_of_scenario
3 - Tag just before merging in the trunk, named scenario-catch_n (can occur zero..n times increase n each time)
4 - Tag when work on scenario has been finished, name scenario-end

Large picture[5]

To describe the process we'll use an example of a (group of) developer(s) developing a GUI for the permission administration in Xaraya.

Let's start with the normal development cycle. Each development task is described in a scenario and assigned a name (([perm_gui]). When the developer(s) are ready to start developing this scenario a tag is created in the trunk ([perm_gui-start]) and a branch for the development effort is created ([perm_gui-branch]). Developers check out a working copy of this branch and start developing.

After a while the developers feel that the scenario is finished and want to have it integrated into the project. For this to happen they start with so called "catch-up" with the trunk, to merge in other approved scenarios. The scenario developers resolve all interoperability problems with their scenario and the others. The "catch-up" with the trunk might be necessary more than once, depending on approval of other scenarios.

Let's assume they have resolved the conflicts and want to finish the scenario. They tag their branch with the ([perm_gui-end]). At this point, the formal work of the developers for this scenario is temporary completed. At the tag-point a working installation of Xaraya is left for others to review and test for approval.

When tests are finished, the scenario branch can be merged back into the trunk and the integrators make sure the trunk is updated to a stable state. Just before merging a tag ([merge_of-perm_gui]) is created in the trunk.

If tests fail and it is necessary to continue development on the scenario the ([perm_gui-end]) tag is removed and scenario developers continue their work. This cycle is repeated as necessary to ensure that the trunk always receives a working system state. It is the responsibility of the integrator to leave the trunk in a working state after integrating a finished scenario. You see that the process always ends up with a working software package, whether it be in a scenario branch, the trunk or a release branch. After closing up, a working system is left behind at all times. This helps the team isolating problems and steering the project on feature inclusion instead of time. The project is at all times in such a state that a release branch can be started.

During the normal development cycle multiple scenarios may be active at any given moment. Obviously the "scenario scheme" will be chosen intelligently as to minimise the number of conflicts expected. Crucial in this respect is communication between different scenario developers. Scenario developers are free to merge in code from other active scenarios if they see good reason to do so. In practice try to avoid this situation. It is better to finish one of the scenarios first, get it approved, do a catch-up with the trunk and then finish the second

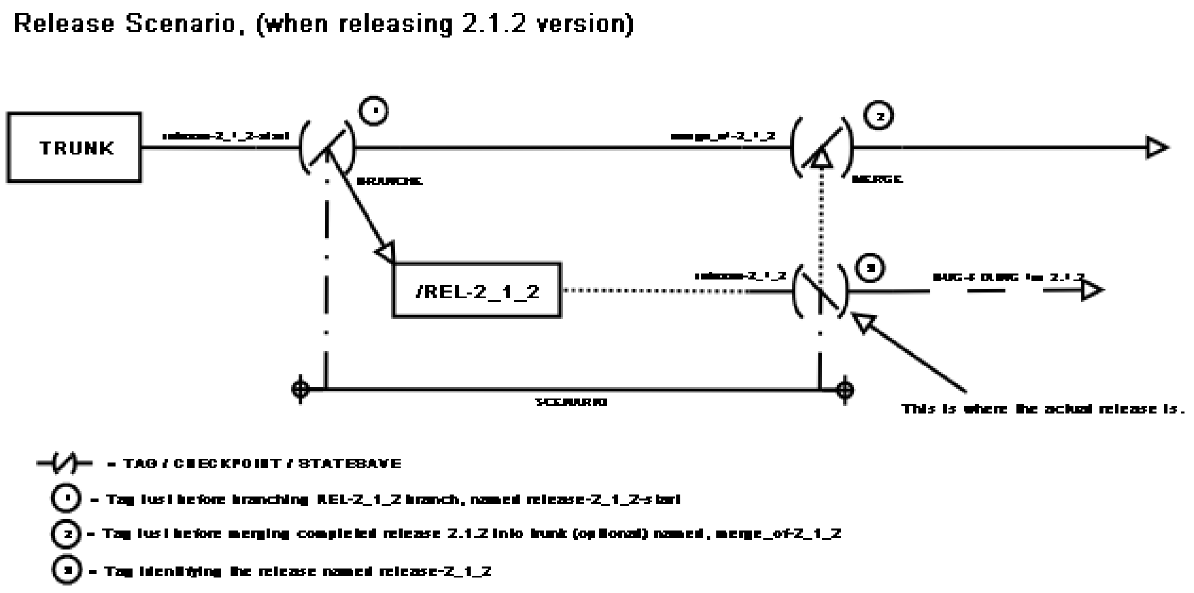
[5] images/rfc0028_1.png

scenario.

The scenario branch is closed for development after a successful merge and may even be removed if appropriate.

1.2.2.2 Release scenario

When it is time to construct a release of the software the following process is used.


Release Scenario, (when releasing 2.1.2 version)
TRUNK
release-2_1_2-start
1
BRANCHING
merge_of-2_1_2
2
MERGE
/REL-2_1_2
release-2_1_2
3
BUG-FIXING for 2.1.2
SCENARIO
This is where the actual release is.
= TAG / CHECKPOINT / STATESAVE
1 - Tag just before branching REL-2_1_2 branch, named release-2_1_2-start
2 - Tag just before merging completed release 2.1.2 into trunk (optional) named, merge_of-2_1_2
3 - Tag identifying the release named release-2_1_2

Large picture[6]

The release scenario isn't that much different from a normal development cycle. When we think a release is warranted a tag is created in the trunk ([release_2_1_2-start]) and a release branch is created ([rel-2_1_2]). Note that this is the same procedure as any normal development scenario, so far.

The first difference is that in a release branch, the development of new features is forbidden. It is a preparation for release and not a new development effort.

Second, no "catch-ups" occurs in a release branch. In the ([rel-2_1_2]) branch the release people start preparing the code for release, in this case release 2.1.2. No merging of the trunk will happen in the branch. The test team can check out a copy of the release branch to do formal testing. When all is ready, the release manager approves the creation of a tag in the branch named ([release-2_1_2]). The state of the tree as it appears when the tag is created is the release. Any change after the tag is there is considered a bug fix after release.

The distribution activities operate on the tree tagged with the ([rel-2_1_2]) tag. Actual distribution activities take place outside the tree, unless they can be done partially before the tag is created. In that case they are considered as release preparation and need to be tested as rigorously as all other features of a release.

The release branch continues to exist after release, but only bug-fixing is done on the branch. At no point in time on a release branch is a merge done from any other branch including the trunk.

Possible support by SCM:

- access privileges to prevent commits in wrong branches.

[6] images/rfc0028_3.png

- automatic regression testing on scenario-branches and trunk Action: warn/cancel commit (depending on how this will be implemented)
- implement multiple stage commit (like review/integrate/commit cycle) Action: depending on SCM

From developers it is expected they can dream the above process.

1.3 Handling commits

The basic action for changing the repository is a commit of one or more changes made to files in the repository. While the development process assures that commits are isolated to a scenario, some guidelines how to manage your commits are in order.

A commit is forever! Period. That's why commits must not be done without any thinking up front. Even if you decide something is wrong and a commit must be rolled back, history shows that it has been done. A rollback doesn't exist; it consists of (at least) two commits.

1.3.1 Where to commit

The development process creates at least three tags for each scenario and at least 2 for each release cycle. Make absolutely sure you are committing your work to the right branch/tag combination. Triple check it! If things are unclear, ask one of the repository managers for help. Backup your own stuff. Quadruple check your commits if you are working on multiple scenarios at once. The SCM tool will support this by setting the appropriate permissions on the scenarios you are working on.

1.3.2 When to commit

All work is done on a scenario branch and the work you commit is limited to that branch. The scenario developers can work out a commit policy. A review will take place at the end of the scenario before the scenario is merged back into the trunk.

1.3.3 What to commit

Honour the scenario scheme! Don't start working on the great new feature in a release branch! If you want to work on that feature, that is ok, describe the scenario, get it approved and a new scenario will be created from the trunk to work on that great new feature.

1.3.4 How to commit

Each commit requires a log message. Use this log feature intelligently. The SCM tool might enforce you to enter certain information: (for example a reference to a test procedure). This will be determined as needed. At the time of writing the log message is free for you to fill with useful information. Why you made a change is a good question to answer in a log message for a commit.

Possible support by SCM:

- use templates for the log messages Action: none
- check content of log messages for minimal content. Action: defer commit
- Integration with tracker facility Action: scan log message / enter tickets directly in repository (investigate this!)

2. Repository tool usage

The second part of this document describes how we implement the repository policy described in the first part with our tool of choice: Bitkeeper[7] [BK]. Everything in this part is BK specific.

2.1 BitKeeper philosophy

If you are used to working with CVS, forget what you know about it. BK works different in most respects although the actual commands used to work with repositories are very much like their CVS counterparts. This part will be easier to understand if you don't translate what is said here to a CVS world.

Probably the biggest thing to get used to is that with BK developers are always working with multiple repositories which have parent-child relationships with each other. Typically you clone a parent repository to a child repository, start working for a while and submit a changeset to the parent repos. At any given moment you may pull changes from a parent to a child or push changes from a child to a parent. If you've read part A of this RFC you will notice that a changeset corresponds in a very intuitive way to a scenario and a pull can be seen equivalent to a catch-up.

Bitkeeper will keep track of all changesets for you. When you try to push a changeset into a parent repository while other changeset have been pushed while you were developing it will tell you to do a pull first to get synchronised with the parent repository. Again, a inherent quality assurance check will be done automatically for you, to prevent pushing changes in the wrong order.

2.2 The overall process

In a BK world you typically clone a repository once and it will live a very long time from that time onwards. BK provides very powerful commands to modify your local repository to synchronise it with parent repositories or peer repositories.

Once you have cloned the parent repository you are in total control how to proceed. You may work directly in the child repository or clone this local repository again and start working in the grandchild, thus using the child as your own stable repository. It's up to each developer how to organise his/her work.

Bitkeeper is very good at merging changes. In nearly all cases it will automatically merge all changes right, even when your repository hasn't been pushed for a long time. (We will dig into this some more later on).

For describing how to use BK in practice we have two situations:

1. Developing as a single developer
2. Developing with a group of developers

Note that even if you work in a group you always work in or with your own repository and that if you work alone the description given for the group can equally well be applied when you are working alone. The alone and group descriptions are used to clarify the process rather than describe your ability to make friends with people.

2.2.1 Developing on your own (direct child of Xaraya)

The following steps describe in short how you would work when rolling your own repository:

1. Clone the Xaraya repository:

```
                    bk clone
user@xaraya.com:/usr/local/repositories/xaraya/core/stable [local-name]
```

   This creates a local repository with its parent set to the xaraya repository at the host xaraya.com identified with the specified user.

2. Make changes to the code in whatever way you are used to. You are automatically working in a scenario

[7] http://www.bitkeeper.com

now. No further work on your part is necessary for this. Note that this is best though of as an implicit scenario. It's not named, nor are the tags explicitly created, the process is preserved however, your changes are local to you and not directly merged into a central repository.

3. When you are finished with the changes to the scenario you can check in your changes. Note that your are checking in your changes to your own repository and not to the parent repository you originally cloned from.
   ```
                       bk ci
                       bk commit
                       or
                       bk citool (GUI)
   ```
   BK will ask you for comments on the changes you made. We recommend using the citool command first as this will guide you through the construction of the changeset. The construction of a changeset is finishing up your scenario and making sensible comments about it. Within the citool you can commit the changeset. Note that this step can be repeated for every scenario you want to make. It allows for intuitive division of work into logical steps, please make sure that you do. Note also that this can all be done off-line on the beach in Florida. No contact is needed with the parent repository.
4. After a number of changesets you can decide to push your changes to the parent repository, after which they can be made visible to other developers.
   ```
                       bk push
   ```
5. If during your development other changesets (new scenarios, bug fixes etc.) have been pushed to the xaraya repository BK will notify you of that and will force you to do a bk pull first.
   ```
                       bk pull
   ```
   Bitkeeper will notify you if there are conflicts and aid you in resolving them with a graphical merge tool. After all conflicts have been resolved, push your change set to the parent repository again. BK will resolve far more conflicts automatically for you than CVS does. For example, when files have been moved, CVS will be stymified, but bitkeeper will still merge your changes to the right place, even if your repository is way behind in directory and file reorganisation (can you say NS-module?)
6. For how to add, delete and move files and directories have a look at the survival guide.

2.2.2 Working in groups (grandchildren of xaraya)

Essentially the same things apply as for working as an individual. The main difference is: If you are working within a group the parent repository may or may not live on xaraya.com. Say developers A,B and C are working on 'blocklayout', coordinated by A.

1. A super-scenario (i.e. a new repository) 'blocklayout' is created on xaraya.com by the repository managers (by cloning the xaraya repository in a special way)
2. All developers who work on that scenario clone (an extra) local repository from xaraya.com:
   ```
                       bk clone
   user@xaraya.com:/usr/local/repositories/xaraya/core/scenarios/blocklayout
   ```
   Note the different name of the repository
3. The same steps as in Developing on your own apply. Of course you should take care to coordinate development somehow. Even the best tool cannot replace that. During the group development of the super-scenario your parent is the blocklayout repository and not the normal xaraya repository.
4. When work on the blocklayout super-scenario is finished, the changesets in the special blocklayout repository can be pushed by the repository managers at once to the xaraya repository. (which is the parent of blocklayout) The same rules apply here for pulling and pushing. In the case of a super-scenario, it is sometimes (depending on the scenario topic) advisable to let the repository managers regularly do a pull from the xaraya repository which the developers can pull to their local repository.

Alternatively:

1. Developer A clones the xaraya repository from xaraya.com for work on the scenario.\
2. A's clone becomes the parent repository. B and C clone their repository from A's machine:

```
                  bk clone user@A.somewhere:/path/to/repository
```

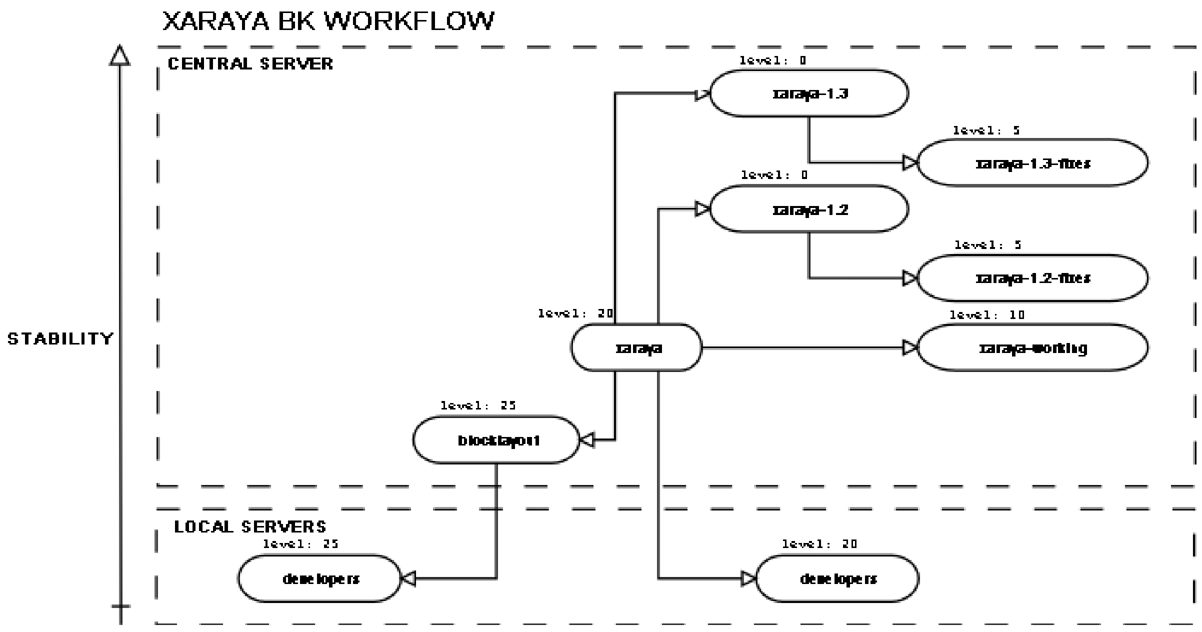
   It is A's responsibility to ensure a secure setup: A has to create ssh accounts for B and C on his machine.

3. The same steps as in Working with your own repository apply.
4. When work on the scenario is finished, A pushes the changesets from his/her repository to the parent: bk push.

Please note: BitKeeper supports a star topology. It is possible to change parents for your repository. We strongly discourage that except for very special occasions, as it tends to make development and conflict resolution much harder. It is easy to get confused, too. To give an example where re-parenting makes sense:

1. A wishes to leave the scenario and work on something else. B takes over.
2. All change sets are pushed to A's repository so that the repositories of A,B and C are in sync. (This is technically not necessary but it does make life easier.)
3. B sets the parent of his repository to the development repository on xaraya.com:
   ```
                     bk parent user@xaraya.com:/usr/local/repositories/xaraya
   ```
4. C sets the parent of her repository to B:
   ```
                     bk parent user@B.somewhere:/path/to/repository
   ```
5. B has to create an ssh account for C on that machine.

Graphically the above works out as follows:



2.3 BK survival guide

We advise you to print out the BitKeeper reference card. It is located in your local installation directory and called bk_refcard.ps or bk_refcard.pdf.

BitKeeper has an integrated graphical, hyperlinked help tool which can be called with the command:

```
            bk helptool
```

A quick help for each command is available with:

```
            bk help [command]
```

What is below as a quick-quick-reference for survival

2.3.1 Commonly used commands

bk pull: Get updates from the parent repository.

bk push: Push change sets to the parent repository.

bk mv [source] [destination]: Move file/dir, recording the move

bk new [files]: Add file(s) to the repository

bk rm [files]: Remove file(s) (recoverable)

bk rmdir [dir]: Remove directory (recoverable)

bk status: Show status information

bk grep [pattern] [files]: Search some/all revisions of file(s) for a pattern

bk delta [file]: check in a change to a file to the local repository, to be added to change sets later on. This can (but need not) be used prior to bk citool: it allows you to enter descriptions for the changes made to single files, in addition to the description the whole change set gets anyway.

bk ci: check in the all changes in current dir into the repository

bk commit: create a changeset from the pending deltas

bk revtool [filename]: GUI for browsing changesets

bk difftool [left] [right]: Graphical diff tool

bk csettool: Graphical change set browser

In summary, bk delta and bk ci register changes to files, bk commit groups them into a changeset, bk push publishes those changesets to the parent repository.

2.4 Use cases

This section will describe some use cases, sometimes made up, sometimes shamelessly copied from other documents on the internet, sometimes real author experience. The examples are meant both as well, err examples and to show that bitkeeper can solve real problems

2.4.1 Bug hunting

Bug fixing can vary from making a quick fix from totally overhauling a set of files to overcome a design flaw.

Suppose you found a bug in the session management code. Apparently it has been introduced a long time ago (it used to work) and you are not sure when it was introduced. You try some versions released earlier but you cannot figure out how or what.

Using bk revtool session.php you quickly glance over all changes made to the file in the past. You find a couple of suspicious changesets based on their comments. Playing with the revtool narrows it down to a changeset which put a brace in the wrong place. You fix the bug in a new changeset, refer in your comment to the old changeset and commit your changes.

If the bug were introduced in a changeset and was isolated, you could have reverted that changeset without touching anything else in the repository, only revoking that particular change

2.4.2 Creating a patch

While working on the 300% speed increase scenario, you need the exception handler code from another developer group. The code is in the tree with several other unrelated changes that they don't want to send to anyone.

Run bk revtool to find the changeset which introduced the exception handler code. Then run

```
                bk export -tpatch -r1.234  > exceptionhandler.diff
```

If the changeset properly grouped the relevant changes this patch can be applied to another tree which needs those changes.

2.4.3 Sharing changes

If another group is working on some changes, but they are not ready yet to be pushed to the xaraya repository, how do you prevent redundant work, so your development can continue?

If the other developer (group) has made their repository publicly available you don't even need the help of the other group. You create the patch yourself by cloning their repository. In most cases different developers are working with the same parent (the default xaraya repository) and don't even need to make a patch. They can do a sideways pull from their repository to incorporate the changes.

2.4.4 Returning from the beach from florida ;-)

You have tanned in Florida for three months without internet access, because your psychiatrist has told you to do so. He didn't know you were using bitkeeper, so you happily developed a great new thing. Returning from Florida you notice that 50 changesets have been pushed while you were away. First, you think about returning to Florida immediately with the

```
                find . -name *.rej
```

in the back of your head, but you give it a try anyway.

bk pull downloads and applies all changes which you missed, regardless how long you have been away. Your repository will never get corrupted by bitkeepers data-integrity checking at every step. You notices that most of the changes are auto-merged with yours, even the complete directory organization you did on the module system. The conflicts which come up are easily resolved using the 3-way merge tool.

2.4.5 New functionality

Developing new functionality is probably the most common activity. For this, a child of the normal xaraya repository must be cloned in which the changesets can be developed. If the new scenario is big enough a central child may be created which may function as a parent for the specific scenario itself. (See Working in groups)

2.4.6 Releasing version 1.2

The PMC decides it's time to release. From the xaraya repository a special clone is made (xaraya-1.2) to which only reviewed changesets can be pushed, typically last minute fixes. When the release is ready the repository is cloned another time (xaraya-1.2-bugfixes) to have a bug-fix repository for the release. The xaraya-1.2 repository is closed for all changes, but can be pulled for changesets from other repositories. Bugfixing for the release occurs in the bug-fixing child xaraya-1.2-bugfixes.

When it is necessary to release a version 1.21 for example the whole process is repeated only with different names. After time some repositories will eventually disappear.

If you think bitkeeper is disk hungry that way, you are right. By using hard-links this can be reduced significantly. If you're in a windows world this is not possible, but that is hardly a surprise.

2.5 Conclusion

The described process is the backbone of our development progress. Before this process was accepted it has been reviewed extensively and voted upon by all relevant people. This process will not change lightly (although improvements will always be looked for).

As you may have noticed, bitkeeper is far more advanced than CVS. This is meant to make our life easier. The repository managers have tried to configure bitkeeper in such a way that it is most easy to use, not necessarily the most efficient way. In time we will probably change some configuration items of our repository management. This RFC (both part A and B will be maintained by the repository managers, to always reflect the latest state of things.

3. Revision history

2003-07-26: updated paths to reflect new repository organisation, some minor fixes

2002-11-27: converted private documents to RFC, small corrections and review, no major changes from original

before 2002-11-27: Private documents