

KMyMoney Project Handbook

Version 1.2

Dipl.Ing. Thomas Baumgart <ipwizard@users.sourceforge.net>

Michael Edwardes <mte@users.sourceforge.net>

Copyright © 2001, 2002, 2005, 2008 Thomas Baumgart, Michael Edwardes, Alvaro Soliverez

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover texts, and with no Back-Cover Texts. A copy of the license is included in the appendix entitled "GNU Free Documentation License".

As for any software development project, certain rules regulate the development process of **KMyMoney**. These rules cover things like coding standards, configuration management and error reporting, just to name a few. The focus of this document is on configuration management and coding standards. More information about the **KMyMoney** project can be found on the project's homepage.

If you have any comments to raise about this document please send an email to <kmymoney2-developer@lists.sourceforge.net>, the project's developer mailing list and we will try to rectify it. Please note that the most up-to-date version of this document can be found in the source tree and is online on the project's homepage. A PDF version is also available for download.

A prerequisite for this document is a basic understanding on the work with **CVS**. Even though this document covers some of the more special topics of **CVS** in more detail it is not intended as an introduction to **CVS** in general.

Table of Contents

History of this document

1. Configuration Management

Version Control Tool

Access to the repository

Read-Only access

Read-Write access

Version controlled files

Files that must be stored in the repository

Files that should not be stored in the repository

Version management

Layout of the version numbers

2. Release Management

Creating a new source version

Creating a new stable version

Announce a new version

Announce new version via File Release System

- Update information about release on web-sites
- Announce new version on mailing lists
- Creating a new binary/installable version
 - Creating an RPM file
 - Copying the tar-ball to the RPM structure
 - Test building
 - Setting up the SPEC file
 - Building the package
 - Testing the package
 - Signing the package
- 3. Coding Rules
 - General
 - Header Files
 - Source Files
- 4. Creating dialog boxes in KMyMoney
 - Language
 - Naming the dialog
 - Designing the dialog
 - Naming widgets
 - i18n considerations
 - Saving the UI
 - Writing code
 - Header (.h) file
 - Code (.cpp) file
 - Updating the Makefile
- 5. Settings
 - How to create a settings page
 - How to add the setting items
 - References
 - Hints
- 6. Unit Testing
 - Why unit testing?
 - Unit testing in **KMyMoney**
 - Unit testing HOWTO
 - Integration of CPPUNIT into the **KMyMoney** project
 - Naming conventions
 - Necessary include files
 - Accessing private members
 - Standard methods for each testcase
 - CPPUNIT_ASSERT
 - CPPUNIT_FAIL
 - CPPUNIT_TEST_SUITE
 - CPPUNIT_TEST_SUITE_END
 - CPPUNIT_TEST
 - CPPUNIT_TEST_SUITE_REGISTRATION
- 7. Documentation
 - General
 - Style Guide

- Tools
- Style Guide
- Producing Final Documents
 - HTML
 - PDF
 - Man Pages (UNIX only)
- 8. Patch Submissions
 - Steps to create a patch
 - Prerequisites
 - If you modified existing Files
 - If you have added new Files
 - Final Steps
- 9. Translation
 - Basics
 - How to get your po file
 - Translating
 - Test your work
 - Merging an old po file with an updated pot file
- 10. Problem Management
 - Reporting problems
 - Referencing problems
 - Problem attributes
 - Reported By
 - Severity level
 - Area
 - Assignee
 - Status
 - Resolution
- A. CVS examples
 - Checking out from the repository
 - Checking in to the repository
 - Updateing changes performed by other developers
 - Dismissing changes
 - Keeping different branches on the same machine
 - Promoting bug-fixes to the main branch
 - Creating a new stable release
- B. Source and Header Examples
 - Header File Example
 - Source File Example
- C. Unit Test Examples
 - Unit Test Header File Example
 - Unit Test Source File Example
 - Unit Test Container Source File Example
- D. RPM SPEC file example
- E. Licence
 - Free Documentation Licence

List of Tables

- 1.1. Definition of version control related terms
- 10.1. Available problem status values
- 10.2. Available problem resolution values

List of Examples

- 2.1. Revisions on the head of a stable branch
- 3.1. Using include stoppers
- 3.2. Including other header files
- 3.3. Class declaration
- 3.4. Complete class declaration
- 3.5. Declaration of slot and signal methods
- 3.6. Attribute naming convention
- 3.7. Including header files in source files
- 3.8. Method implementation
- 3.9. Kernighan & Ritchie flow control style
- 3.10. Allman flow control style
- 3.11. One line body flow control style
- 3.12. Local variable naming convention
- 7.1. Using include stoppers
- A.1. Filling the sandbox for the first time
- A.2. Filling the sandbox for the first time with a specific version
- A.3. Promote changes to the repository
- A.4. Updating the sandbox
- A.5. Checking the status of files in the sandbox
- A.6. Reverting changes made to the sandbox
- A.7. Keeping stable and development branch on one machine
- A.8. Checking out the stable branch for the first time
- A.9. Promoting a change from the release to the development branch
- A.10. Creating a new stable branch

History of this document

This chapter contains a list of releases of this document. Each entry in this list contains a date, an author and a short description and possibly a release number.

2001-11, Thomas Baumgart, Rev 0.1

- Initial work on the project handbook goes back into November 2001. Unfortunately, the only history information is available in **CVS**.

2005-09-18, Thomas Baumgart, Rev 1.0

- Added chapter about translation provided by J. Rundholz.
- Added this version history

2006-09-11, Thomas Baumgart, Rev 1.1

- Added chapter about documentation provided by Tom Browder.

2008-01-31, Alvaro Soliverez, Rev 1.2

- Added chapter about submissions
- Added chapter about settings

Chapter 1. Configuration Management

Table of Contents

- Version Control Tool
- Access to the repository
 - Read-Only access
 - Read-Write access
- Version controlled files
 - Files that must be stored in the repository
 - Files that should not be stored in the repository
- Version management
 - Layout of the version numbers

Whenever a project is developed in stages, it is very important for the people working on the project to know, which version of the project is used by the users and how it was built and what the components of this project are. This applies to hardware (e.g. automobiles) as well as software. Think of the times, when a car manufacturer calls certain cars to be repaired due to production problems. The manufacturer can track down the relevant cars because he is using a configuration management system and knows each car that contains e.g. faulty parts.

The same applies to software development projects: whenever a version (stage of development) of the project is made public, it has to be given a distinct name or tag - usually the version number - and the development team must keep a snapshot of all components that made up that stage of the software project. This does not only include the source code of the project, but all libraries in their relevant stage as well as compilers and tools and all their installation/configuration information. If all this information is present, the software development team is capable to exactly reproduce each delivered version to search for problems occurring with this version. If this capability is not available, the development team has a hard time to find these problems.

CVS helps us in our project to cover one aspect of configuration management: version control of the source code. It helps us to keep the snapshots of all the files the project team is responsible for in a central repository. Another aspect of CVS is concurrent development which allows us that some of us can develop new features of the project while others fix problems with previous versions even if both tasks modify the same source file. This feature is called branching. How branches will be organized in our project is explained in the chapter about Version management.

The configuration management also regulates how the development team passes information among the members. This includes things like naming conventions, how errors are reported, rated and fixed and who is responsible for which task. The emphasis on this document though is the management of the version control system and how things are handled in this area. This does not mean, that the other important issues of configuration management are left outside. They are just not in the focus of this document.

Throughout this document a few terms are used. In order to avoid confusion because these terms might be used differently in other documents, they are defined here.

Table 1.1. Definition of version control related terms

Term	Definition
Revision	A <i>revision</i> is the stage of a single file in the repository. <i>Revisions</i> start with the value 1.1 upon the initial check-in and are incremented with each check-in of that file. After the third check-in, the <i>revision</i> of a file has the numeric value of 1.3. Once branches are made, <i>revisions</i> can have values like 1.6.2.2. Since the <i>revisions</i> differ for all files, the revision number is only necessary for certain administrative tasks on single files (e.g. merging data from another branch).
Tag, Label	A <i>tag</i> is a string that represents a single revision of a file. Multiple tags can point to the same revision of a file. Sometimes, <i>label</i> is used as a synonym for tag.
Version	A <i>version</i> is the stage of the whole project in the repository. As already mentioned, the revisions for the files contained in the project differ from each other for a specific <i>version</i> . Therefore, each revision contained in a <i>version</i> receives the same tag. This tag can be used to checkout an exact copy of the version of the project.
Repository	The repository is the central database containing all revisions of all files of the KMyMoney project. It is located on the SourceForge.net and can be accessed via CVS .
Sandbox	The sandbox is the local work area for the developer. Initially, a sandbox is filled by checking out a specific stage of the repository. Changes made to the sandbox are moved to the repository by the checkin process. Changes made by other developers are transferred to one's own sandbox by the update process. A developer can maintain different sandboxes of the same project in different directories on the same machine. This requires thorough attention of the developer which sandbox he is modifying. Using several sandboxes is usually meaningful if a developer works on changes on a release branch (stable release) and on the main-branch (development release) at the same time. See the appendix for an example.

Version Control Tool

Since the **KMyMoney** project is hosted on the SourceForge platform, CVS is used as the version control tool. **CVS** is widely accepted as version control tool in the open source community and covers our needs pretty well.

The **KMyMoney** project's central repository is handled on the SourceForge platform. Developers do not edit files directly in the repository, but rather checkout a working copy into their local sandbox. This local sandbox can then be modified without the necessity of a direct link to the central repository. Once the developer is confident with the changes made, he checks the files back into repository.

During the checkin process, **CVS** keeps track of all the changes made. This allows to review a complete history of all modifications ever made to the project. As mentioned above, it is very important in certain circumstances to have such a history.

Access to the repository

Access to the repository is available in two different forms

- Read-Only access
- Read-Write access

Read-Only access

Read-Only access is granted to anybody anonymously to the repository. For this type of access you do not need a user account on the SourceForge platform. Access is made through the *pserver*-protocol of **CVS**: See the description on SourceForge for more details.

Read-Write access

In order to get read-write access to the repository, two things are necessary. First you need a user account on the SourceForge platform. Second you need to qualify as a developer with the **KMyMoney** project administrator(s). Once they have added you to the list of developers, you can access the repository through an *SSH* encrypted tunnel. See the description on SourceForge for more details.

Version controlled files

This chapter explains which files have to be version controlled. It also explains how to configure **CVS** to skip certain files that are generated by the compile/build process but should not be stored in the repository.

Files that must be stored in the repository

All files that form an application (e.g. source code, header files, icons, documentation, etc.) that are necessary to compile, build and run the application **must** be checked into the central repository.

Caution

Before you add a file to the repository you must check it's format. If it's a binary format the special option **-kb** must be appended to the **cvs add** in order to inform the central repository to leave the file as it is

Files that should not be stored in the repository

All files that are automatically generated by the build process (e.g. object files, libraries, executables, Makefiles, etc.) should not be checked into the repository because they can easily be reconstructed by the developer. In very rare circumstances it might be necessary to checkin an automatically generated file. This is always an exception.

Note

I mentioned Makefiles above because I assume that **autoconf** and **automake** are used. Using **autoconf** and **automake** supplies the project with a **configure** script that creates the Makefiles. The necessary input files are called **Makefile.am** that must be checked into the repository as source to the Makefiles.

If you do not use **autoconf** and **automake** and write your Makefiles directly, they have to be checked in as they are not automatically generated. Nevertheless, using non **automake** generated Makefiles should be avoided.

Version management

At certain times to be defined by the project's administrators (actually configuration manager) a snapshot is taken from the repository. In order to fix the stage of this snapshot in the repository, a *tag* is placed on this stage. Tagging the repository creates a version of the project.

Layout of the version numbers

The version number is made out of three numeric fields. These are:

1. The major release number
2. The minor release number
3. The micro release number

In order to serve as a tag for **CVS**, the text **rel-** is prepended to the version numbers and the three fields are separated with a dash (e.g. rel-0-3-7).

The major release number will be changed when all the proposed features mentioned in the release plan (available on the project web-site) are designed, coded and tested.

One of the objectives of the minor release number is to differentiate between stable and unstable versions of the project. As a widely accepted procedure, odd numbers are used for development versions, even numbers identify stable versions. Whenever the configuration manager decides that to create a new stable release, this will also bump the version number of the unstable release by two.

The micro release number is reset to 0 when the minor release number is incremented and then incremented with each version following until the minor release number is modified again. Versions tagged between the creation of a stable branch and the actual release -0 of this branch will be identified by a micro release number of pre_n , where n is incremented each time a version is tagged.

Chapter 2. Release Management

Table of Contents

Creating a new source version
Creating a new stable version
Announce a new version

- Announce new version via File Release System
- Update information about release on web-sites
- Announce new version on mailing lists
- Creating a new binary/installable version
 - Creating an RPM file
 - Copying the tar-ball to the RPM structure
 - Test building
 - Setting up the SPEC file
 - Building the package
 - Testing the package
 - Signing the package

At certain stages, the development team releases a version of **KMyMoney**. The following chapters explain the steps that are performed during this process.

Creating a new source version

The process of releasing a new version is to build a source tar-ball archive, verify that **KMyMoney** can be built from it and upload it to the SourceForge File Release System. This chapter explains the steps of making sure that the version numbers are set correctly, creating the source tar-ball, tagging the repository etc.

Note that in order to perform some of the functions associated with this procedure, you will need to:

1. Obtain a userid by registering with Sourceforge; this userid is represented below as `<your_sf_userid>`
2. Be registered as a KMyMoney developer; this must be done by a project administrator, the names of which appear on the project home page on Sourceforge.

The first few steps of the release process should be taken some time in advance of the anticipated release date, in order to give translators a chance to 'do their thing'. The length of time required will depend on how many translatable strings have been changed since the previous release, but something like two weeks for a minor release should suffice.

Note

The steps explained apply to both the development and the release branches. For convenience, the examples are based on the development branch.

1. Determine the release number which will identify this release.

Two types of versions can be created at this time: a *follow-up* release or a *fresh stable* release. The follow-up release is based on a previous release with the same major and minor release number. The fresh stable release starts a new major and minor release number pair. In the latter case, the major and minor release numbers on the development branch in CVS are also adjusted. More details below.

- a) For follow-up releases increase the micro-release-number by one since the last unstable/stable release.

- b) For a fresh stable release, increase the minor release number and set the micro-release number to 0.

2. Create a new directory for this release

Create a new directory specifically for the release process, something like /home/me/distkmm, which will ensure that the following steps are not contaminated by other, existing versions of **KMyMoney**. From this directory, check out a copy of the app from the CVS branch which forms the basis of this release, e.g. for 0.8.2, the branch will be rel-0-8-branch. The checkout process will create a sub-directory called kmymoney2. This is referred to subsequently as the TLD (top level directory).

3. Update the text source file for translations.

This is done from the TLD by running the command:

```
tjb: ~> make package-messages
```

This will create the file kmymoney2.pot in TLD/po, and will merge all new and changed messages into the various translation files in the same directory. These files (kmymoney2.pot and *.po) should be committed to the appropriate branch of CVS. Also, an announcement should be made on the translator's mailing list that these are ready for updating, and mentioning a date a few days before the proposed release date as a deadline for translations to be submitted.

At this point also, a 'string freeze' for the base release should be declared on the developer's list. From now on until the release is complete, the only changes which should be committed to the CVS branch should be fixes which do not change translatable strings, and updated .po files submitted by translators.

As each .po file is committed, it is desirable to update the translation statistics on the project web site. This can be performed with the command

```
make message-stats | ssh <your_sf_userid>@user.sourceforge.net "cat > /home/groups/k/km/kmymoney2/htdocs/translate-stable.xml"
```

You should certainly make sure that this command is run at least once, after all .po files have been committed.

Once the anticipated release date is reached, you should make sure that your sandbox is up-to-date. This is probably best achieved by deleting the directory created in step 2, and re-creating it by a full CVS checkout as described there.

4. Check that the version number is correct.

In the TLD, check file configure.in.in for the correct version number. A line near the top should look like

```
AM_INIT_AUTOMAKE(kmymoney2, 0.8.3)
```

If the last digits don't match the release number, then change them and commit your change to the repository with the message 'Bumped to release x.y.z'.

5. Verify the libtool version number

Visit all subdirectories that contain a shared library. In the TLD, issue the command

```
find . -name Makefile.am -exec grep -H LIBVERSION= '{}' \;
```

(Note that the space before the backslash is necessary.) For each match, check whether the code or interface of the shared library has been changed since the last release. If so, modify the LIBVERSION setting in each Makefile.am according to the following recipe, where the LIBVERSION setting controls the libtool versioning system for shared libraries. It is constructed out of the triplet CURRENT:REVISION:AGE.

- a) If the library source code has changed at all since the last release, then increment REVISION ('C:R:A' becomes 'C:r+1:A').
- b) If any interfaces have been added, removed, or changed since the last update, increment CURRENT, and set REVISION to 0.
- c) If any interfaces have been added since the last public release, then increment AGE.
- d) If any interfaces have been removed since the last public release, then set AGE to 0.

Caution

Make sure to modify the numbers in coordination with changes on the release- and development branch. It could well be, that the REVISION must be incremented more than 1 if it has been changed on the other branch as well.

6. Perform an initial build of the application

From the TLD, issue the commands:

```
thb: ~> make -f Makefile.dist release
thb: ~> ./configure <any-options-you-need>
thb: ~> make
```

This will regenerate all files of the application and rebuild everything automatically.

7. Check the distribution

Before doing this, check that all desktop files conform to the freedesktop specification. From the TLD, issue the following command:

```
thb: ~> find ./kmy money2 -name '*.desktop' -exec desktop-file-validate {} \;
```

N.B. Any errors in file x-kmy money2.desktop can be ignored since kde3 does not conform to the specification for this file type.

Note: desktop-file-validate is part of the desktop-file-utils package, which may be installed from your distribution repository, or downloaded from <http://www.freedesktop.org/wiki/Software/desktop-file-utils>

Then, to check the distribution itself, from the TLD, run the command

```
thb: ~> make distcheck
```

This will do the following things automatically and stop on any error:

- a) create a source tar-ball in tgz form
- b) unpack this source tar-ball in a separate directory
- c) run configure on the unpacked source tar-ball
- d) compile and link the configured package
- e) compile and link all testcases
- f) install the compiled program in a temp directory
- g) check that all files are installed
- h) uninstall the package from the temp space
- i) check that no files are left behind

Make sure that everything builds correctly. If errors occur, correct them and rerun **make distcheck**. Once everything is ok, a respective message, that the tar-ball is ready for distribution is shown at the end of **make distcheck**.

8. Update the ChangeLog

Add a line with the text

```
* Released x.y.z
```

to the ChangeLog file.

9. Commit changes to CVS

Commit the updated ChangeLog file, and any Makefiles updated in step 5 to the CVS repository before you proceed with the next step. Use the message 'Released x.y.z'.

10. Tag the version on CVS

From the TLD, issue the command 'cvs tag rel-x-y-z', where x-y-z is the release number with dots replaced by dashes.

11. Update and optimize the size of the tar-ball

As the content has changed since the tar-ball was created in step 7, we need to re-create it. Create a more compressed version using the command:

```
thb: ~> make dist-bzip2
```

12. Make a checksum of the tarball

This can be done with the following command from the TLD

```
thb: ~> md5sum name-of-tarball.tar.bz2 >/home/me/somewheresafe
```

This checksum may be used to verify downloaded files at a later stage, and may be added to sourceforge at some future time.

13. Create a Release Note and a release ChangeLog file.

The former should contain a brief description of the release and any new features which have been added. The latter should be a tidied-up version of the application ChangeLog file, though any purely internal changes may be omitted.

14. Upload the tarball to Sourceforge's 'incoming' directory

Upload the tar-ball (bz2-version) to

ftp://anonymous:<your-email-address>@upload.sourceforge.net:incoming, making sure to use the binary transfer mode.

For GUI users: *anonymous* is the user-name and *your-email-address* is the password.

For command line FTP, from the TLD:

```
thb: ftp
ftp> open upload.sourceforge.net
Connected to osdn.dl.sourceforge.net.
<snip>
Name (upload.sourceforge.net:tonyb): anonymous
331 Please specify the password.
Password:<your_sf_userid>
230 Login successful.
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> cd /incoming
250 Directory successfully changed.
ftp> binary
200 Switching to Binary mode.
ftp> put name-of-tarball.tar.bz2
ftp> quit
```

15. Move the tarball to the Sourceforge File Release System

Next pull the uploaded file into the **KMyMoney** section of the File Release System on SourceForge so that the file will be visible to everyone on the internet. To do that, load the following URL in your browser

```
https://sourceforge.net/project/admin/editpackages.php?group_id=4708
```

At the bottom of the page, click on the 'Edit releases' link. If the release (x.y.z) you've built doesn't appear in the list, go back a page and click on the 'Add release' link to add it, then return to 'Edit releases', then click the 'Edit this release' link for your release.

In Step 1 of the page, set the status to Hidden for now, paste the Release Notes and ChangeLog into the appropriate boxes and Submit.

In Step 2, select the kmymoney tarball file and Add.

In Step 3, set Processor to 'platform-independent', File Type to 'source .bz2', and Submit.

16. Activate tarball on Sourceforge

Once you are happy that the tarball was uploaded okay, and the release announcements are all correct, enter the File Release system again, and set the status to Active.

17. Update bug lists

If the ChangeLog indicates that any Sourceforge or KDE bugs have been fixed in this release, log on to the respective bug sites and ensure that they are marked as closed.

18. Announce the release

(At this point, you may wish to wait a few hours to allow Sourceforge to populate its mirror sites, thus avoiding complaints to the mailing lists.) Announce the presence of the source tar-ball archive as described in Announce new version.

19. Prepare for next release

Make sure that you increase the project version to the next version. This is derived as follows:

- a) After a follow-up release, increase the micro-release-number by one, e.g. if the release you are currently working on is called 0.7.3 then set the release number to 0.7.4.
- b) After a fresh stable release, increase the minor release number and set the micro-release number to 0, e.g. if the release you are currently working on is called 0.6 then set the release number to 0.7.0.

Make the appropriate changes to configure.in.in as described in step 4 above. From the TLD, issue the commands:

```
thb: ~> make -f Makefile.dist
thb: ~> ./configure <any-options-you-need>
thb: ~> make
```

This will regenerate all files of the application and rebuild everything.

Finally, check in the updated configure.in.in to the CVS repository.

Note

The version number in the sandbox is *always* the version number that is currently being developed (we're a little ahead of ourselves here).

Creating a new stable version

At a certain time in the project's development cycle, the configuration manager decides that a feature freeze is necessary to start a new stable version. The exact dates when this will happen are announced on the developers mailing list ahead of the event. When the time has come to freeze the features, a branch will be created as described in this section. From this time on, the stable release will only be

changed to make the current features of the software more stable. New features are not introduced to the stable branch but can be developed on the main branch (unstable) in parallel.

When the time has come to create a new stable branch, the following steps have to be performed.

1. Run through all the steps explained in Creating a new version. The version number used in this description for the stable version is 0.4. Follow the path for a fresh stable release.
2. Create a branch off of the tagged version. The branch name is build by appending the word *-branch* to the major- and minor-release number of the stable release version. For our example, the branch tag for versions 0.4 is *rel-0-4-branch*. A complete example with all **CVS** commands can be found in the appendix.
3. From this moment on, the developers working on versions 0.4.x must make sure, that they checkout or update their sandbox using the tag *rel-0-4-branch*. This gives them the head revisions of the files on the 0.4 branch. Omitting this tag information will leave them on the main branch. The main branch is reserved for the unstable versions. An example how to keep multiple branches on the same machine is presented in the appendix,

Caution

The developers really have to take care from this point on which version they are modifying in their sandbox. Besides that, it is the developers responsibility to make sure that bug-fixes are also implemented on the main-branch if applicable.

When fixes are applied to the branch, new versions can be created by incrementing the micro-release thus *rel-0-4-1*, *rel-0-4-2* are the next tags on the release branch.

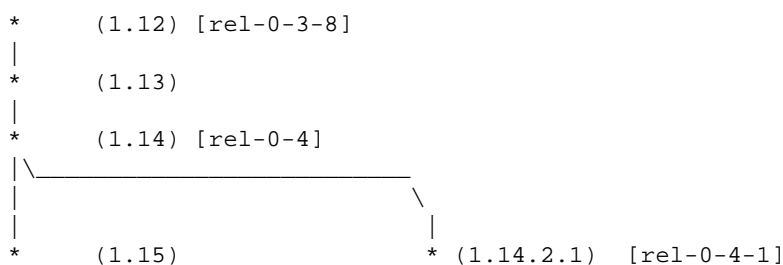
Note

Since **CVS** does not allow periods inside a tag, we always replace periods (.) with dashes (-) inside a tag.

The following diagram shows the above example on two specific files. Each node represented by an asterisk is labelled with it's revision number enclosed in parenthesis. If a node has one or more labels attached, then they are enclosed in brackets. Nodes may exist without a tag. Such revisions never went into a release neither stable nor unstable but are valid intermediate steps in the development of the file in question.

Example 2.1. Revisions on the head of a stable branch

The first file is changed rather often between the version tags. All tags are on different revisions of the file.



Announce new version on mailing lists

Write a short mail and send it to the developer- and user-mailing list of the project, so that all subscribed recipients are informed about the new release. Add links to the project web-site and the www.kde-apps.org page of the project.

Creating a new binary/installable version

Additionally, installable binary versions of **KMyMoney** should be provided. Since the installable binary files differ from distribution to distribution and the generation in general requires the target platform, the **KMyMoney** project relies on the help of people not directly involved in the application development. We greatly appreciate any help in this area.

Multiple formats exist: RPM, DEB, e-builds, PKG just to name a few. Since the distro I use (SuSE) relies on RPMs, I explain the creation in more detail here. If you can provide similar information about other formats, we are more than happy to include it in this document. We assume that you follow our licence terms for any documentation you supply. Please send your docbook formatted files to the developer mailing list.

Creating an RPM file

One possibility to distribute the program is to use the Red-Hat Package manager (RPM) format. In order to be able to create such a package for **KMyMoney**, you need to have a source tar-ball as described in the previous chapter.

The RPM system uses a directory structure which for my system - a SuSE distribution - is located in `/usr/src/packages`. This may be different on your system. The location can be configured in `/etc/rpmrc`. I will refer to this directory as the 'RPM base directory' in the remainder of this document.

The RPM base directory has a set of subdirectories. They all serve a specific purpose. For us, the directories `SOURCES`, `SPECS`, `SRPMS` and `RPMS` are important. `RPMS` is further divided into directories for specific CPU architectures (e.g. `i386`, `i486`, `ppc`, etc.). In the remainder of this document, I will use the names of these directories without mentioning the RPM base directory.

Copying the tar-ball to the RPM structure

The first thing that needs to be done is to copy the tar-ball to a defined place where the RPM tool will look for it. For this purpose the `SOURCES` directory is used. Move or copy your tar-ball to this directory.

Test building

The first thing you'll probably want to do is get the source to build cleanly without using RPM. To do this, unpack the sources, and change the directory name to `$NAME.orig`. Then unpack the source again. Use this source to build from. Go into the source directory and follow the instructions to build it. If you have to edit things, you'll need a patch. Once you get it to build, clean the source directory. Make sure and remove any files that get made from a configure script. Then `cd` back out of the source directory to its parent. Then you'll do something like:

```
thb:~> diff -uNr dirname.orig dirname > ../SOURCES/dirname-distrname.patch
```

This will create a patch for you that you can use in your spec file. Note that the "distro-name" that you see in the patch name is just an identifier. You might want to use something more descriptive like "MDK9" or "RPM8" to describe why you had to make a patch. It's also a good idea to look at the patch file you are creating before using it to make sure no binaries were included by accident.

Note

This section has been copied from the RPM-Howto and adapted where necessary

Setting up the SPEC file

The next step is to create an RPM SPEC file for the specific distribution. The contents may vary between distribution and that is where your knowledge is required. An example for an RPM SPEC file is contained in appendix ???. It will work on SuSE 8.1 directly. More details on howto setup a SPEC file including an explanation of the various sections, commands and options is contained in the RPM-Howto.

Building the package

Once you have the spec file it's time to try and build your package. The usual way to do this is using the following command:

```
thb:~> rpmbuild -ba kmymoney.spec
```

Once the command finishes successfully, you have a source RPM in SRPMS and a binary RPM for your distribution in one of the subdirectories of RPMS.

Note

More details about this process and a description on the available options can be found in the RPM-Howto.

Testing the package

Once you have a source and binary rpm for your package, you need to test it. The easiest and best way is to use a totally different machine from the one you are building on to test. After all, you've just done a lot of make install's on your own machine, so it should be installed fairly well.

You can do an `rpm -e packagename` on the package to test, but that can be deceiving because in building the package, you did a make install. If you left something out of your file list, it will not get uninstalled. You'll then reinstall the binary package and your system will be complete again, but your rpm still isn't. Make sure and keep in mind that just because you do a `rpm -ba package`, most people installing your package will just be doing the `rpm -i package`. Make sure you don't do anything in the build or install sections that will need to be done when the binaries are installed by themselves.

Note

This section has been copied from the RPM-Howto and adapted where necessary

Signing the package

Once you are confident with the RPM package, it is a good idea to sign it with your secret **GPG** key before you distribute it. Signing the package allows any recipient of the package to verify that it has not been altered by an unauthorized party.

Signing will create a separate file that contains the electronic signature for the RPM file. In order to allow any recipient to verify the signature, two things have to be kept in mind:

- Always distribute both files together
- Make your public key available on e.g. <http://www.keyserver.net>.

The following example shows the command sequence necessary to create an ASCII armored signature.

```
thb:~> gpg -b -a kmymoney2-0.5.1-1.i386.rpm
You need a passphrase to unlock the secret key for
user: "Thomas Baumgart <thb@net-bembel.de>"
1024-bit DSA key, ID B75DD3BA, created 2001-06-23

Enter passphrase: I WON'T TELL YOU MY PASSPHRASE ;- )
thb:~>
```

Once you have entered the correct passphrase the signature file will be created under the name `kmymoney2-0.5.1-1.i386.rpm.asc`. As an example, I include it here. **THIS IS NOT THE REAL SIGNATURE, EVEN IF IT LOOKS LIKE IT.**

```
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.0.7 (GNU/Linux)

iD8DBQA+E1DInFnbQLdd07oRAMFQAKDV0I9nzxGEIh1Mx/tzoZ4J3Iyt6gCfTXl1
LrISXXgD6xELWZlO+NswbLw=
=qJIP
-----END PGP SIGNATURE-----
```

These two files, the RPM and the signature, should be distributed to the public. The receiver of these two files can now verify if the RPM file is the one you signed or has been modified. Therefore, he needs your public key which he gets from one of the public key servers (e.g. <http://www.keyserver.net>) into his keyring. The verification is performed using **GPG** as the following example shows:

```
thb:~> gpg --verify kmymoney2-0.5.1-1.i386.rpm.asc
gpg: Signature made Wed 01 Jan 2003 09:16:37 PM CET using DSA key ID B75DD3BA
gpg: Good signature from "Thomas Baumgart <thb@net-bembel.de>"
thb:~>
```

Note

Besides signing the RPM package, the SRPM (Source-RPM) package should be signed as well.

Chapter 3. Coding Rules

Table of Contents

General
Header Files
Source Files

Where-ever possible this document should be referred to when questions regarding the format of the source code are raised.

By the way, we know the code doesn't always conform to the standards at the moment, but work is underway to change the code and all new code submitted should conform to the standards.

General

The following list shows the general rules that should be regarded by any developer working on **KMyMoney**.

- Each file should contain only one declaration or implementation and the filename should reflect the class name. e.g `ksomethingdlg.h` would contain a declaration for the `KSomethingDlg` class.
- A tab width of 2 spaces should be used and if your editor supports it, the tabs should be changed into spaces. (KDevelop/KWrite supports tab translation).
- All dialogs should be located in the `kmymoney2/dialogs` directory.
- Each class should be as self contained as possible. If for instance, you are creating a dialog, then all the signals and slots should be connected within that dialog class. Where access is needed to the class details methods should be used. This enhances readability and makes maintenance a lot easier with each object having it's own state, identity and behaviour, (see Object Oriented Analysis & Design using UML, Bennet & Co).
- All user visible text should be wrapped in the `i18n` internationalisation wrapper for translation.

Header Files

The following rules apply to all header files.

- Header files shall end with the extension `.h` not `.hpp`.
- All header files shall begin with a comment block as generated by KDevelop.
- The remainder of the header file shall be surrounded by include stoppers. The name of the macro used should be the capitalized filename with the dot replaced by an underbar (e.g. `KSettingsDlg.h` --> `KSETTINGSDLG_H`)

Example 3.1. Using include stoppers

```
#ifndef KSETTINGSDLG_H
#define KSETTINGSDLG_H
    /* remainder of header file */
#endif // KSETTINGSDLG_H
```

- All classes designed for use by the KDE interface should begin with a *K* with each separate word beginning with an uppercase letter e.g KSomethingDlg.
- The header file will include other header files in the following fashion and same order:

Example 3.2. Including other header files

```
//-----
// QT Headers
#include <qtlabel.h>

//-----
// KDE Headers
#include <kcombobox.h>

//-----
// Project Headers
#include "mymoney/mymoneyfile.h"
```

- Each class should have a kdoc compatible comment header to describe the class and it's function within kmymoney2.
- Classes shall begin their declaration as such:

Example 3.3. Class declaration

```
class KSomethingDlg : public KBaseClass {
```

with an appropriate access declared.

- Access modifiers should be left flushed in the class declaration with all attributes and methods indented by one tab. The access methods will be in order starting with private. The access identifier should exist even if no attributes or methods exist. Only one identifier can exist of the same type.

Example 3.4. Complete class declaration

```
class KSomethingDlg : public KBaseClass {
private:
    QString m_szSomeString;
    void useString(void);

private slots:

protected:

protected slots:
```

```

public:
    KSomethingDlg();

public slots:

signals:
};

```

- All slot methods should begin with slot and signal methods should start with signal. e.g

Example 3.5. Declaration of slot and signal methods

```

signalFoundTransaction();
slotFoundTransaction();

```

- Attribute names should begin with the `m_` prefix to indicate that they are member variables. The variable name should begin with a descriptive identifier such as `qcomboboxMethod`. Explicit hungarian notation is also fine. Examples of valid variable names can be found below:

Example 3.6. Attribute naming convention

```

QComboBox m_qcomboboxMethod;
int m_intCounter;
int m_nCounter;

```

- Method names should specify a return and argument(s) unless used in a slot or signal where the argument list can be left blank if necessary. The method should start with a lower case letter with each subsequent word having an upper case start letter.

Source Files

The following rules apply to all source code files.

- C++ source files shall end with the extension `.cpp` not `.cc` or `.cxx`
- As with header files these should start with a header block similar to the one generated by KDevelop.
- Include files shall be included in the same format as for header file e.g

Example 3.7. Including header files in source files

```

//-----
// QT Headers
#include <qtlabel.h>

//-----
// KDE Headers
#include <kcombobox.h>

//-----
// Project Headers
#include "mymoney/mymoneyfile.h"
#include "ksomethingdlg.h"

```

- Methods should be implemented as such:

Example 3.8. Method implementation

```
void KSomethingDlg::useString(void)
{
    .. function body
}
```

with the function body indented by one tab (equals two spaces).

- Flow control statements should preferably follow the Kernighan & Ritchie style as such:

Example 3.9. Kernighan & Ritchie flow control style

```
while (something_is_true) {
    operate on something;
}
```

although the following Allman style is acceptable:

Example 3.10. Allman flow control style

```
while (something_is_true)
{
    operate on something;
}
```

It is also acceptable for one line body statements to omit the curly braces as such:

Example 3.11. One line body flow control style

```
while (something_is_true)
    operate;
```

- Local variables should not be prefixed by the m_ member prefix and should start with a prefix as discussed for the header file. For example:

Example 3.12. Local variable nameing convention

```
QString qstringTemp;
char *pszTemp;
```

- Each method should have a comment block preceeding it in a suitable format for other developers to see how the method works and what types of return and arguments it expects. It does not have to be kdoc compatible because kdoc only parses the header files. All kdoc comment blocks should be in the header files.

Chapter 4. Creating dialog boxes in KMyMoney

Table of Contents

- Language
- Naming the dialog
- Designing the dialog
 - Naming widgets
 - i18n considerations
 - Saving the UI
- Writing code
 - Header (.h) file
 - Code (.cpp) file
- Updating the Makefile

This section is a developer's guide explaining the peculiarities of dialog creation in KMyMoney. A basic understanding of Qt GUI programming is assumed, as is a knowledge of KMyMoney coding standards as laid out in the Project Handbook.

Language

The default language of KMyMoney is American English, but don't let that put you off! All contributions will be welcome, and as long as you have a basic knowledge of the language, the members of the KMyMoney developers list will be happy to help you polish it up.

Naming the dialog

First step is to choose a meaningful name for your dialog (and preferably one that's not already in use! - see the `kmymoney2/dialogs` source directory). In accordance with KMyMoney coding conventions, the name should consist of 2 or 3 (or more) 'words', strung together without spaces. To keep names to a manageable length, these 'words' may be abbreviated, e.g. 'sched' for 'schedule'. Each 'word' should be spelt with an initial upper-case letter. Also, the names of dialogs are always preceded by a letter 'K', thus e.g. `KEditSchedTrans`. This name will be indicated in the rest of this chapter by `<KN>`. There are nevertheless occasions where all lower case letters are appropriate, specifically source file names; this will be indicated as `<kn>`.

Designing the dialog

The dialog screen should be built using Qt Designer. This section assumes that you are using version 3.x of Qt. Version 4 is, at present, an unknown quantity.

Open Designer without specifying a project, and select Dialog from the New File/Project tab. Start by changing the form name; this should be set to '`<KN>DlgDecl`'.

Now add your widgets to the form, not forgetting to include a Help button. Remember that users will have many different hardware and screen combinations, and will need to be able to resize windows, so make full use of the various layout and spacer options of Designer. A lot of tutorials can be found on the web to help guide you on this; try your favourite search engine.

Naming widgets

Fixed widgets, e.g. text labels, can often use the default names assigned by Designer. Other widgets on your form should be given names which are meaningful in an application context. This is particularly important for those widgets which are to be referenced in code. As per the application programming standards, these names should be prefixed with 'm_' to indicate them as member variables of the dialog.

i18n considerations

Designer contains an option to generate shortcut (accelerator) keys for various widgets (buttons, menu items) by including an ampersand ('&') before the shortcut letter. This should be used for the more common items, since many users prefer to use keyboard input rather than using the mouse. However, this does have the unfortunate side effect of automatically generating an 'accel' property for the widget, referencing a letter which may not be appropriate when the caption is translated to another language. Use the properties menu, therefore, to remove this value, or see below.

Fixed text fields and labels in the form do not require any special consideration. Qt Designer and the project's build environment will take care of wrapping the strings into an i18n construct for presentation to translators.

Saving the UI

When complete, save the form using the Designer default name (<kn>dlgdecl.ui) in the dialogs source code folder (kmy money2/dialogs).

Writing code

Your code to process form actions should be included in source files named <kn>dlg.h/.cpp, in the same folder as the .ui file. You can view these for many examples of how to code. Some requirements are:

Header (.h) file

This should start with definitions similar to the following

```
#ifndef <KN>DLG_H
#define <KN>DLG_H

#include "../dialogs/<kn>dlgdecl.h"

class <KN>Dlg : public <KN>DlgDecl {
    Q_OBJECT
public:
    <KN>Dlg(QWidget *parent = 0, const char *name = 0);
    ~<KN>Dlg();
};
```

The first two lines are the standard include stoppers, to avoid multiple inclusion of the class data.

The include file will have been generated by the Qt UIC (User Interface Compiler) from the .ui file for the dialog, under control of the make process.

The `Q_OBJECT` macro (written without any punctuation) will cause the Qt MOC (Meta Object Compiler) to generate additional object code and files which are necessary to support the signal/slot functionality (among other things).

The class declaration must also include a

```
public slots:
```

and

```
signals:
```

sections if you plan to use the signal/slot mechanism. See the Qt documentation about signals and slots. An example would be `slotHelp()` which will be connected to the `clicked()` signal of the help button of your dialog in the constructor of your dialog.

Terminate the file with

```
#endif
```

to close off the include stoppers.

Code (.cpp) file

First, don't forget to have `#include` directives for Qt headers for any widgets you are going to reference.

In the constructor function, connect all signals to their appropriate slots using the Qt `connect()` call.

Then the easy bit; write your code.

Finally, terminate the source file with the following

```
#include "<KN>dlg.moc"
```

This is one of the files generated by the Qt MOC (Meta Object Compiler) during the make process; if you finish up with 'vtable' errors, it's probably because you forgot to include this.

Updating the Makefile

You will need to edit file `Makefile.am` in the dialogs source folder before building `KMyMoney`. Note that due to the abstruse rules of `make`, the lists of files should consist of a single logical line, so be careful regarding any editor options which may cause automatic insertion of line breaks. You can however use a continuation character of backslash to spread the list over multiple physical lines. There must be no character following the continuation character, not even a blank.

- Add `<kn>dlgdecl.ui` and `<kn>dlg.cpp` to the `libdialogs_a_SOURCES` line
- Add `<kn>dlgdecl.ui` to `EXTRA_DIST`
- Add `<kn>dlgdecl.cpp` and `<kn>dlgdecl.h` to `DISTCLEANFILES`

- Add `<kn>dlg.h` to `NOINST_HEADERS`

Save the file, and you are ready to build KMyMoney. For the first build after updating `Makefile.am` you should re-run `'make -f Makefile.dist'`, reconfigure and make. Otherwise, some make rules might not be present and compiling fails.

That's all, simple wasn't it.

Chapter 5. Settings

Alvaro Soliveres <asoliveres@gmail.com>

Table of Contents

How to create a settings page

How to add the setting items

References

Hints

How to create a settings page

- Create the view using designer, name it `XxxDecl` and store it in `kmymoney2/dialogs/settings/xxxdecl.ui`. See more information about naming the items further down
- Create the class that contains the logic for the settings page, name it `Xxx` and store it in `kmymoney2/dialogs/settings/xxx.[cpp|h]`.

Don't forget the `Q_OBJECT` macro at the beginning of the class declaration in the `.h` file and make the class a public derivative of `XxxDecl`

- Add the `xxxdecl.ui` and `xxx.cpp` filename to the `libsettings_a_SOURCES` label in `kmymoney2/dialogs/settings/Makefile.am`
- Add the `xxxdecl.ui` filename to the `EXTRA_DIST` label in `kmymoney2/dialogs/settings/Makefile.am`
- Add the `xxxdecl.cpp` and `xxxdecl.h` filename to the `DISTCLEANFILES` label in `kmymoney2/dialogs/settings/Makefile.am`
- Add the `xxx.h` filename to the `noinst_HEADERS` label in `kmymoney2/dialogs/settings/Makefile.am`
- Add the construction code to `KMyMoney2App::slotSettings()` as

```
Xxx* xxxPage = new Xxx();  
dlg->addPage(xxxPage, i18n("text"), "icon-name");
```

where you replace "text" with a short text that shows up under the icon in the settings view and "icon-name" with the name of the icon for that settings page

- Make sure to include `xxx.h` in `kmymoney/kmymoney2.cpp`

How to add the setting items

For auto-generation of setter/getter code of your options, you have to follow certain rules. For each setting item you need an entry in `kmymoney2/kmymoney2.kcfg`. This is an XML formatted file. The contents of the 'name' attribute will be used as method for the C++-code, eg. a name of "AutoSavePeriod" for an integer parameter results in a setter and getter named

```
void KMyMoneySettings::setAutoSavePeriod(int)
int KMyMoneySettings::autoSavePeriod(void)
```

You should not access those functions directly from within your code but rather use the `KMyMoneyGlobalSettings` class which contains the same interface as `KMyMoneySettings` with some additional functionality.

When you name the GUI widget that controls the setting for this parameter make sure to name it "kcfg_AutoSavePeriod", that is "kcfg_" prepended with the name used in `kmymoney2/kmymoney2.kcfg`.

That should be it.

References

A more complete - but generic version - can be found on http://techbase.kde.org/Development/Tutorials/Using_KConfig_XT. Currently, **KMyMoney** does not use the `CMakeLists.txt` file but the above mentioned `Makefile.am` approach. You can safely skip the section about `CMakeLists.txt`.

Hints

If you run a `make` 'too early' it could be, that certain entries for the `Makefile` are not setup correctly and the compiler will complain. In this case, try to run a `make -f Makefile.dist` followed by `./configure` and see if the problem goes away.

Chapter 6. Unit Testing

Table of Contents

Why unit testing?

Unit testing in **KMyMoney**

Unit testing HOWTO

- Integration of CPPUNIT into the **KMyMoney** project

- Naming conventions

- Necessary include files

- Accessing private members

- Standard methods for each testcase

- CPPUNIT_ASSERT

```
CPPUNIT_FAIL
CPPUNIT_TEST_SUITE
CPPUNIT_TEST_SUITE_END
CPPUNIT_TEST
CPPUNIT_TEST_SUITE_REGISTRATION
```

If this project handbook would have been for a professional project (with professional I mean, a project that people make money with), I would have written

Caution

Unit tests must be supplied by the developer with the classes/source-code he checks into the repository!.

Since this is the handbook for a voluntary work (which is not less professional than any other project), I replace the above sentence with

Note

Each developer in this project is strongly encouraged to develop unit tests for the code he or she develops and make them available to the project team!.

Why unit testing?

Before I can give an answer to this question, I should explain what unit testing is about. I do not cover all relevant aspects here nor do I start a discussion of the various aspects of unit testing. If you want to read more about the details of unit testing, the philosophy behind it and about the various tools available, please visit the project pages of JUnit and CPPUNIT. The following explanation describes what unit testing is:

For each class developed in a project, an accompanying test container is developed when the interface of the class is defined but before the implementation of the class starts. The test container consists out of testcases that perform all necessary tests on the class while verifying the results. One or more of these test containers (for more than one class) form a test suite.

Your might think, that it is strange to first define the interface, then develop the tests and then start the development of the actual code, but it has shown, that this approach has a couple of interesting side-effects:

- The developer spends time to think about how to test his implementation before he actually works on the implementation. This leads to the fact, that while working on the implementation, he already knows how his code will be tested.
- A clear definition of the *end of implementation* exists due to the fact, that the testcases will all fail before the beginning of the implementation phase. Once implementation proceeds, more and more testcases will pass. When they all pass, the development is finished.
- Since the tests will run automated and will be re-run very often during the development cycle, a lot of problems will be caught very early on. This reduces the number of problems found during integration of the project. Believe me, there will be plenty left!

Now, the list of all these side-effects is actually the answer to the question *Why unit testing?* or does anyone have a argument against it? I agree, that in some cases automated unit testing is hard to achieve (e.g. for GUI related code) but I found, that whenever it is possible to introduce automated unit tests, the benefit is huge.

Unit testing in KMyMoney

Just about the time, when the **KMyMoney** project underwent a radical change of it's inner business logic (the KMyMoney engine), I read an article about the existance of a unit test container for C++ projects named CPPUNIT. In discussions with my colleagues at work, I got the impression, that this would be something worth to look into. So I sat down and wrote the first test cases for existing code to get a feeling for what is required.

I found it annoying to write test cases for code that existed and was believed to work (version 0.4 of the project). When the decision was made to start with the 0.5 development branch, I started working on the new engine code that should introduce a clear interface between the business logic and the user interface. Another design goal was to write the engine in such a way, that it is not based on any KDE code which the old one was. The second goal to free it from Qt based code was not that easy and was skipped by the project team at that time.

Even if it was hard for me at first to follow the above laid out principle to design the interface, write the test code and then start with the implementation, I followed this trail. It has proven to be very valuable. Once the interface was designed, I started thinking in a different manner: How can I get this class to fail? What strange things could I do to the code from the outside? Those were the design drivers for the test code. And in fact, this thinking changed the way I actually implemented the code, as I knew there was something that would check all these things over and over again automatically.

A lot of code was implemented and when I was almost done with the first shot of the implementation, discussion came up on the developers mailing list about a feature called *double entry accounting* that was requested for **KMyMoney** by a few people. The engine I wrote up to that point in time did not cover the aspects of double entry accounting at all, though a few things matched. After some time of discussions, we became a better understanding of the matter and I changed the code to cover double entry accounting. Some of the classes remained as they were, others had to be adopted and yet others rewritten entirely. The testcode had to be changed as well due to the change in the interfaces, but not the logic of the tests. Most of the thoughts how to uncover flaws remained.

And that is another reason, why unit testing is so useful: You can change your internal implementation and still get a feeling, if your code is working or not. And believe me: even if some changes are small, one usually oversees a little side-effect here and there. If one has good unit tests this is not a problem anymore, as those side-effects will be uncovered and tested.

During the course of implementing the engine, I wrote more than 100 testcases. Each testcase sets up a testenvironment for the class and tests various parameters against the class' methods in this environment in so called test steps. Exceptions are also tested to be thrown. The testcases handle unexpected exceptions as well as expected exceptions that do not occur.

Unit testing HOWTO

This section of the developer handbook should give some examples on how to organize test cases and how to setup a test environment.

My examples will all be based on the code of the **KMyMoney** engine found in the subdirectory **kmymoney2/kmymoney2/mymoney** and it's subdirectory **storage**. A single executable exists that contains all the test code for the engine. It's called **autotest** and resides in the **mymoney** subdirectory.

Integration of CPPUNIT into the KMyMoney project

The information included in the following is based on version 1.8.0 of CPPUNIT. The **KMyMoney** build system has been enhanced to check for it's presence. Certain definitions setup by *automake/configure* allow to compile the project without unit testing support.

Caution

This is not the recommended way for developers!

If code within test environments is specific to the presence of CPPUNIT it can be included in the following `#ifdef` primitive:

```
#ifdef HAVE_LIBCPPUNIT
// specific code that should only be compiled,
// if CPPUNIT >= 1.8.0 is present
#endif
```

For an example see the Unit Test Container Source File Example.

The same applies for directives that are used in **Makefile.am** files. The primitive to be used there is as follows:

```
if CPPUNIT

# include automake-directives here, that should be evaluated
# only, when CPPUNIT is present

else

# include automake directives here, that should be evaluated
# only, when CPPUNIT is not present.

endif
```

For an example see **kmymoney2/mymoney/Makefile.am**.

Naming conventions

The test containers are also classes. Throughout CPPUNIT, the test containers are referred to as *test fixtures*. In the following, I use both terms. For a given class *MyMoneyAbc*, which resides in the files **mymoneyabc.h** and **mymoneyabc.cpp**, the test container is named *MyMoneyAbcTest* and resides in the files **mymoneyabctest.h** and **mymoneyabctest.cpp** in the same directory. The test container must

be derived publicly from **CppUnit::TestFixture**. Each testcase is given a descriptive name (e.g. EmptyConstructor) and I found it useful to prefix this name with the literal 'test' resulting into something like testEmptyConstructor.

Necessary include files

In order to use the functionality provided by CPPUNIT, one has to include some information provided with CPPUNIT in the test environment. This is done with the following include primitive as one of the first things in the header file of the test case container (e.g. mymoneyabctest.h):

```
#include <cppunit/extensions/HelperMacros.h>
```

Accessing private members

For the verification process it is sometimes necessary to look at some internal states of the object under test. Usually, all this information is declared private in the class and only accessible through setter and getter methods. Cases exist, where these methods are not implemented on purpose and thus accessing the information from the test container is not possible.

Various mechanisms have been developed all with pros and cons. Throughout the test containers I wrote, I used the method of redefining the specifier *private* through *public* but only for the time when reading the header file of the object under test. This can easily be done by the C++ preprocessor. The following example shows how to do this:

```
#define private public
#include "mymoneyabc.h"
#undef private
```

The same applies to protected members. Just add a line containing *#define protected public* before including the class definition and a line containing *#undef protected* right after the inclusion line.

Standard methods for each testcase

Three methods must exist for each test fixture. These are a default constructor, setUp and tearDown. I think, it is not necessary to explain the default constructor here. setUp and tearDown have a special function within the test cases. setUp() will be called before the execution of any test case in the test fixture. tearDown() will be called after the execution of the test case, no matter if the test case passes or fails. Thus setUp() is used to perform initialization necessary for each test case in the fixture and tearDown() is used to clean things up. setUp() and tearDown() should be written in such a way, that all objects created through a test case should be removed by tearDown(), i.e. the environment is restored exactly to the state it was before the call to setUp().

Note

This is not always the case within the testcase for **KMyMoney**. Especially when using a database as the permanent storage things have to be overhauled for e.g. MyMoneyFileTest.

CPPUNIT comes with a set of macros that help writing testcases. I cover them here briefly. If you wish a more detailed description, please visit the CPPUNIT project homepage.

CPPUNIT_ASSERT

This is the macro used at most throughout the test cases. It checks, that a given assumption is true. If it is not, the test case fails and a respective message will be printed at the end of the testrun.

CPPUNIT_ASSERT has a single argument which is a boolean expression. The expression must be true in order to pass the test. If it is false, the test case fails and no more code of the test case is executed. The following example shows how the macro is used:

```
int a, b;
a = 0, b = 1;
CPPUNIT_ASSERT(a != b);
a = 1;
CPPUNIT_ASSERT(a == b);
```

The example shows, how two test steps are combined. One checks the inequality of two integers, one the equality of them. If either one does not work, the test case fails.

See the Unit Test Source File Example for a demonstration of it's use.

CPPUNIT_FAIL

This is the macro used when the execution of a test case reaches a point it should not. This usually happens, if exceptions are thrown or not thrown.

CPPUNIT_FAIL has a single argument which is the error message to be displayed. The following example shows how the macro is used:

```
int a = 1, b = 0;
try {
    a = a / b;
    CPPUNIT_FAIL("Expected exception missing!");
} catch (exception *e) {
    delete e;
}

try {
    a = a / a;
} catch (exception *e) {
    delete e;
    CPPUNIT_FAIL("Unexpected exception!");
}
```

The example shows, how two test steps are combined. One checks the occurrence of an exception, the other one that no exception is thrown. If either one does not work, the test case fails.

CPPUNIT_TEST_SUITE

This macro is used as the first thing in the declaration of the test fixture. A single argument is the name of the class for the test fixture. It starts the list of test cases in this fixture defined by the CPPUNIT_TEST macro. The list must be terminated using the CPPUNIT_TEST_SUITE_END macro.

See the Unit Test Header File Example for a demonstration of it's use.

CPPUNIT_TEST_SUITE_END

This macro terminates the list of test cases in a test fixture. It has no arguments.

See the Unit Test Header File Example for a demonstration of it's use.

CPPUNIT_TEST

This macro defines a new test case within a test fixture. As argument it takes the name of the test case.

See the Unit Test Header File Example for a demonstration of it's use.

CPPUNIT_TEST_SUITE_REGISTRATION

This macro registers a test fixture within a test suite. It takes the name of the test fixture as argument.

See the Unit Test Container Source File Example for a demonstration of it's use.

Chapter 7. Documentation

Tom Browder <tom.browder@gmail.com>

Table of Contents

- General
- Style Guide
- Tools
- Style Guide
- Producing Final Documents
 - HTML
 - PDF
 - Man Pages (UNIX only)

Code documentation is discussed in the section "coding." This section discusses developer and user documentation and generating html and pdf versions of same. Note that all non-source-code documentation, with two exceptions, must be written in docbook form.

Note

The two exceptions are `./kmymoney2/html/home.html` and `./kmymoney2/html/whats_new.html` which are used on KMyMoney's internal home page when running the application.

General

In general, all documentation for KMyMoney should follow guidelines for the KDE project. In addition to the KDE guidelines, there are KMyMoney guidelines (which take precedence if there are conflicts). See the following KDE resources:

1. <http://110n.kde.org/>
2. <http://110n.kde.org/docs/markup/index.html>
3. <http://110n.kde.org/docs/tools.php>
4. <http://people.fruitsalad.org/phil/kde>
5. <http://people.fruitsalad.org/phil/kde/pdf-stuff/pdf-instructions.html>

XML entities should be used for commonly used terms and phrases. There is a KMyMoney list at

1. `./developer-doc/phb/kmymoney-entities.docbook`

Style Guide

Tools

1. `meinproc` (used to produce HTML from docbook; part of KDE base)
2. `dblatex` (used to produce PDF from docbook; add-on from KDE doc team; see resources)
3. `??check??` (used to check docbook formatting, etc.; part of KDE base)
4. `??check??` (used to check consistency of word and phrase usage; add-on from KDE doc team; see resources)

Style Guide

Producing Final Documents

HTML

Example 7.1. Using include stoppers

(command)

PDF

Man Pages (UNIX only)

Chapter 8. Patch Submissions

Alvaro Soliverez <asoliverez@gmail.com>

Table of Contents

Steps to create a patch

Prerequisites

If you modified existing Files

If you have added new Files

Final Steps

This section describes how to send patches and additions when you don't have direct CVS access. In that case, you should send the contributions to the developer mailing list. That way, more people can test your contribution than if you send it to a specific developer. It could also happen that this developer has a lot of pending stuff, and your contribution gets delayed.

For the specifics on how to code, translate or write documentation, refer to the proper sections. Once you are done with the actual work, you have to create the patch file to send.

Steps to create a patch

Prerequisites

Have an updated cvs version of the release you are contributing to (HEAD, stable release, etc)

- Make sure to run 'cvs upd' before you create the patch
- If the update changes your sandbox, make sure that the changes still work

If you modified existing Files

- Run 'cvs diff -u' in the root directory. That should create the patch to be applied to existing files
- Inspect the patch that it does only contain your wanted changes

If you have added new Files

- Make sure you write down on the email the location of each new file (The developer handling the patch will probably know how to find the correct location, but this will save her/him some precious time)

Final Steps

- Compress the patch and any new files into a single .tar.gz file
- Send to the developer mailing list, explaining the nature of the submission
- The developer handling the patch should acknowledge that to the list as well, to avoid duplicate work

Keep in mind that there are times when all developers have plenty of work, or a patch should be best handled by a specific developer that may not be readily available, so be patient. Also, if you don't receive an acknowledgement after a few days, write to the list asking for the status of your patch. Subscribing to the developer mailing list would be a good idea if you are sending contributions.

Chapter 9. Translation

J. Rundholz

Table of Contents

Basics

How to get your po file

Translating

Test your work

Merging an old po file with an updated pot file

This chapter should give any one (in particular people who are not developers) a first overview on how they can translate **KMyMoney** into another language. The information given here is not only valid for **KMyMoney** but as serves for the translation of any KDE project.

Basics

First of all you need to get the required files. There are two types of files which might be of interest to you. Files ending in `.pot` and files ending in `.po`.

Files ending in `.pot` are the source for the translation. They only contain the English texts you need to translate and are generated for each release of the project. The one for **KMyMoney** is named `kmymoney2.pot`. The po file contains the English and the translated text, e.g. German. There might be two cases: The first case you can't find a po file for your language or the second there is already a po file for your language available. In the latter case you can already start **KMyMoney** in your language, but may be the translation is not complete or you simply want to improve the translation. Now you need to get these files from the project's **CVS** repository. You don't need to worry what exactly **CVS** does, for a translator it is more or less only a file system from where you can download the files you need. For **KMyMoney** you can either go to the **CVS** page and call the **CVS** web front end or follow this link. I would recommend to download the pot and po file of the latest stable version (at the moment 0.8), not any older files and not the one which is under development at the moment. In order to test your translation later you should run the same version on your box. For the actual translation I recommend the program `kbabel`. Other people use `emacs`, as usual it is just a personal choice. All examples given here are based on `kbabel`.

How to get your po file

As explained earlier the actual translation is done by modifying the po file. The next step is to create a po file to start the translation.

No po file for my language available

That's the most simple case: Rename `kmymoney2.pot` to `lang.po` where *lang* is your language code e.g. *de* for German.

po file for my language available

In case you already have a language file it is advisable that you check how the English text entries in that file differ from the English texts in the pot file. Maybe for some reason the pot file is further developed than the po file. In this case even though you would translate the whole po file you would still see some English texts in **KMyMoney**. If you know that this is not the case, i.e. your po file has all the latest English text inside, you can use this one for translation. The minimum check you should perform is to compare the number of messages. Just load the po and the pot file into `kbabel` and you see how many messages you have in each file.

In case that both files differ you should merge them (since you don't want to lose someone else's work). Merging can be done using `kbabel` very easily .

- Copy `kmymoney2.pot` to `lang-new.po` and open this file with `kbabel`.
- In `kbabel` choose Tools Rough translation from the menu. A dialog pops up. Choose the following options:
 - What to translate::Untranslated Entries
 - Options::Mark changed entries as fuzzy
 - Dictionaries::Use::PO Compendium

Don't select anything else.

- Now press the "configure" button. In the next dialog enter the path to your already existing po file. Furthermore set the options:
 - Options::Case sensitive
 - A text matches if::Equal to searched text

Don't select anything else.

Caution

The original po file could have entries marked as "fuzzy", that means that the translation might be not very good or even horribly wrong. After you did this merger process you lose this information. It might be a good idea to translate all fuzzy strings in the old, unmerged po file before you do this explained merging process.

Translating

Now you can take the new po file and start the translation with `kbabel` or any other tool. It might be a good idea to create a glossary of terms whenever you use a technical term. Such a glossary is available for the German translation, check it out to see how it should look like. You might want to check the `gnucash` project if they have already a glossary for your language. This would be a starting point for you, but please extend it.

Test your work

Last but not least you need to check your translation, especially if the translated text fits into the context. Also you shouldn't be surprised how many typos and mistakes you make while translating the text. Check it carefully! Now a few steps you should perform to test your translation (after you used a spellchecker etc.).

First you can run the command

```
thb:~> msgcat --width=77 -o lang-output.po lang-input.po
```

as root. This command just restricts the length of each line to 77 characters. This is only necessary if you want to read the raw po file in some text editor. Some people who do proof reading prefer this instead of endless lines. The number of characters depends of some personal taste. I prefer about 50 characters per line, then a diff file can be displayed without vertical scrolling.

Now run as root the command

```
thb:~> msgfmt -vvv lang.po kmymoney2.mo
```

The mo file is the language file you need for your program. You just need to replace the original mo file of **KMyMoney** with this one (perform a backup of the original file before). The location of your mo file depends on your distro, for SuSE it is /opt/kde3/share/local/lang/LC_MESSAGES and for Debian /usr/share/local/lang/LC_MESSAGES. For lang you can use any string, like de or even de_test if you want to run it with a test language. Be aware that **KMyMoney** uses some text strings from KDE and since they are probably not available with de_test you still might see some English text. If you can't find the file on your harddrive try either

```
thb:~> find / -name kmymoney2.mo
```

as root or if you used an **rpm** file for the installation you can run

```
thb:~> rpm -q kmymoney2 --filesbypkg
```

In order to run **KMyMoney** either start it as usual (if your default KDE language is the language you want to test) or call it via

```
thb:~> KDE_LANG=de kmymoney2
```

from a konsole or xterm.

Merging an old po file with an updated pot file

There are cases when the pot file is regenerated before you can send your updated po file. If you just commit your file after that, it will show up as old in the translation stats, because it references an outdated pot file.

To fix this, you have to merge the po file with the updated pot file. First, get the latest pot file. And in the po folder, run this command.

```
thb:~/kmymoney2/po> make -f ../admin/Makefile.common package-merge POFILES="de.po" PACKAGE=kmymoney2
```

You can merge multiple po files at the same time. Just list them in the POFILES option.

```
thb:~/kmymoney2/po> make -f ../admin/Makefile.common package-merge POFILES="de.po es.po" PACKAGE=kmymoney2
```

Chapter 10. Problem Management

Table of Contents

Reporting problems

Referencing problems

Problem attributes

- Reported By
- Severity level
- Area
- Assignee
- Status
- Resolution

This chapter is a first draft. It contains some ideas that have to be validated by the developer community.

Reporting problems

Problems (that covers errors, enhancement request, etc.) concerning the **KMyMoney** project are maintained using the SourceForge.net platform. This is the sole location, where problems have to be reported. The source for such a problem report can be one of the developers or any other user of **KMyMoney**.

Referencing problems

Once added to the database, the problem will be assigned a unique problem number. This number must be mentioned whenever the problem is referenced (e.g. in the subject of a mail to the developer mailing-list, a checkin comment or an entry in the ChangeLog file). To allow searches for the number, a specific format has to be used for these references. The format is *#assigned-number*, e.g. #481229.

Problem attributes

Besides the fixed problem number which is created when the report is filed, each problem has a couple of attributes that might change during the life-cycle of the problem. They will be described in the following chapters.

Reported By

As the problem number, this field is fix. It represents the SourceForge username of the individual who filed the report.

Severity level

The SourceForge.net platform allows to assign a severity level to each problem. It ranges from 1 to 10. The meaning in our project and the consequences are defined as follows:

FIXME: A more detailed description of the prios needs to be given

1. Lowest priority
2. TBD!
3. TBD!

4. TBD!
5. Medium priority (default)
6. TBD!
7. TBD!
8. TBD!
9. TBD!
10. Highest priority

Area

TBD!

Assignee

This field represents the SourceForge.net username of the individual working on the problem. Each developer should pick problems he feels competent to work on. The possibility to assign another developer to a problem should be used only to gather a comment from this developer. This should be clearly marked as comment with the report.

Note

The number of unassigned problem reports should be 0 most of the time to give clear signal to the world that the developers are working on the project!

Status

Each problem has a status. Right after filing a report, it will be assigned the status 'open' automatically by the SourceForge.net platform. Whenever a developer is working on the problem, he will have to modify the value of this field to reflect the current status. The following values are available and have the associated meaning:

Table 10.1. Available problem status values

Status	Meaning
open	The problem is not yet fixed. A developer might be working on it.
closed	The problem has been closed. No further action is required.
deleted	???
pending	The problem has been modified solved and feedback from the original poster is required. Pending entries will turn into closed entries after 14 days.

Resolution

The following values are available and have the associated meaning:

Table 10.2. Available problem resolution values

Resolution	Meaning
Accepted	The problem report has been accepted by the developers. Nevertheless, it has not yet been duplicated but from the initial report it could well be a problem with KMyMoney
Duplicated	The problem has been duplicated by one of the developers.
Fixed	The problem has been fixed. The code is available via CVS.
Invalid	The report is not valid. It's not a problem related to KMyMoney .
Later	???
None	???
Out of date	The report is based on an older version of KMyMoney and has been resolved in the meantime in a newer release which is available for download or CVS.
Postponed	The problem has been acknowledged but will be postponed until later. The developer changing the state to Postponed should leave a comment with the entry why it is postponed.
Rejected	The problem has been rejected by the development team. The developer changing the state to Rejected should leave a comment with the entry nameing the reasons for rejection.
Remind	???
Wont fix	???
Works for me	The problem cannot be duplicated but seems to be a valid problem. The entry needs more investigation.

Appendix A. CVS examples

Table of Contents

- Checking out from the repository
- Checking in to the repository
- Updating changes performed by other developers
- Dismissing changes
- Keeping different branches on the same machine
- Promoting bug-fixes to the main branch
- Creating a new stable release

It is not the intention of this document to give a complete introduction to **CVS**. Nevertheless, a few examples should give a quick overview on the basic operations to be performed by the developer.

It is assumed, that the developer is registered with the **KMyMoney** project and has read/write access to the repository. Also, it is assumed, that the necessary environment variables are setup, so that **CVS** knows how to access the repository. Details about the settings can be found on the SourceForge.net web-site.

Throughout the next sections, the examples given use the **CVS** command line interface. The options are abbreviated. The operations described here are also accessible through various GUI clients available for **CVS**. Also, I usually use the `-q` (quiet) option to suppress some messages issued by **CVS**. If you omit the `-q` option, the output differs from the one shown here, even though the result of the operation is the same.

Checking out from the repository

The very first operation is to fill the sandbox. This is done using the **checkout** operation. The first time the repository is checked-out, it's location must be specified. This is done using the `-d` option. In the example below, you must replace *username* with your real username at SourceForge.net.

Example A.1. Filling the sandbox for the first time

```
thb:~> cvs -d username@kymoney2.cvs.sourceforge.net:/cvsroot/kymoney2 co kymoney2
```

During the checkout process, **CVS** lists all filenames on the users screen and stores information about the repository's location and all the files checked out in the sandbox. Therefore, you do not need to specify the repository location during following **CVS** operations anymore.

For the **KMyMoney** project, a directory named **kymoney2** is created in your current working directory.

The above example fills the sandbox with the HEAD revision of all files. This stage is sometimes referred to as the "latest-and-greatest" and is the latest development stage.

Note

If you plan to keep two or more branches of the project on your machine, please see the chapter *Keeping different branches on the same machine* for details.

If for some reason, you need to checkout a version of the project that is different from the development stage (e.g. you want to fix a bug in a stable version), you can fill an empty sandbox by supplying the version-tag as parameter to the checkout command.

Example A.2. Filling the sandbox for the first time with a specific version

```
thb:~> cvs -d username@kymoney2.cvs.sourceforge.net:/cvsroot/kymoney2 co -r version-tag kymoney2
```

This will store the version labelled with the tag *version-tag* in your sandbox. In case *version-tag* is a branch-tag, you are able to modify the files and check-in changes later on. In case, *version-tag* is a standard tag, checkin operations will be rejected by **CVS**.

As in the previous example, the directory `kmymoney2` is created as the sandbox.

Checking in to the repository

Once the sandbox is filled, changes to the project will be applied by the developer. As soon as the developer is confident with the changes, he is about to promote these changes to the other developers. He does that by checking the changes back into the repository.

Checking changes back into the repository should start by performing an update procedure as described in the next section. This may seem strange, but updating your sandbox will transfer changes performed by other developers in the meantime to your sandbox. It is good practice to re-compile the project if you notice that updating the sandbox changes its contents. This assures that the project is still compilable when you check-in your changes.

The next step is to identify the changes you really want to promote. This can be performed by the **diff** operation supported by **CVS**.

Example A.3. Promote changes to the repository

For the following example, I assume a single file that has been changed in the sandbox (`~/kmymoney2/kmymoney2/knewbankdlg.cpp`) and that the current directory is `~/kmymoney2/kmymoney2`. Also, it is assumed, that the file `README` has been updated by another person in the repository. Since the `README` file has no influence on the compile process, we omit recompiling in this example.

The part of the original file that has been changed is shown here to understand the output of the **cv diff** command shown below. The beginning of the file is not included here as it is not changed.

```
void KNewBankDlg::okClicked()
{
    if (nameEdit->text().isEmpty()) {
        KMessageBox::information(this, i18n("The institution name field is empty. Please enter the name."), i18n("Adding New Institution"));
        nameEdit->setFocus();
        return;
    }

    m_name = nameEdit->text();
    m_city = cityEdit->text();
    m_street = streetEdit->text();
    m_postcode = postcodeEdit->text();
    m_telephone = telephoneEdit->text();
    m_managerName = managerEdit->text();
    m_sortCode = sortCodeEdit->text();
    accept();
}
```

The changed version of the method is included here.

```
void KNewBankDlg::okClicked()
{
    if (nameEdit->text().isEmpty()) {
        KMessageBox::information(this, i18n("The institution name field is empty. Please enter the name."), i18n("Adding New Institution"));
        nameEdit->setFocus();

    } else {
        m_name = nameEdit->text();
        m_city = cityEdit->text();
        m_street = streetEdit->text();
        m_postcode = postcodeEdit->text();
        m_telephone = telephoneEdit->text();
        m_managerName = managerEdit->text();
        m_sortCode = sortCodeEdit->text();
        accept();
    }
}
```

Now as the file has been changed, the changes should be promoted to the repository. As explained above, the process starts with checking for changes made by other people.

```
thb:~> cvs -q upd
U README
M knewbankdlg.cpp
thb:~>
```

The above shown output has the following meaning: the file **README** is updated (U) from the repository to the sandbox because it has been changed by someone else in the meantime. The contents of the file in the sandbox will be replaced by the contents of the file in the repository, because it has not been altered in the sandbox. The file **knewbankdlg.cpp** has been modified (M) in the sandbox and needs to be returned to the repository.

As the next step, one should check what has been changed in the file **knewbankdlg.cpp**. This is done using the following command:

```
thb:~> cvs -q diff knewbankdlg.cpp
74,75d73
<     return;
<   }
77,84c75,84
<   m_name = nameEdit->text();
<   m_city = cityEdit->text();
<   m_street = streetEdit->text();
<   m_postcode = postcodeEdit->text();
<   m_telephone = telephoneEdit->text();
<   m_managerName = managerEdit->text();
<   m_sortCode = sortCodeEdit->text();
<   accept();
---
>   } else {
>     m_name = nameEdit->text();
>     m_city = cityEdit->text();
>     m_street = streetEdit->text();
>     m_postcode = postcodeEdit->text();
>     m_telephone = telephoneEdit->text();
>     m_managerName = managerEdit->text();
>     m_sortCode = sortCodeEdit->text();
>     accept();
>   }
thb:~>
```

The output shows the changes between the current and the original revision of the file. If this is what needs to be changed then the next step can be started, which is checking the changes back into the repository.

```
thb:~> cvs -q ci -m "Avoid return in the middle of a function" knewbankdlg.cpp
Checking in knewbankdlg.cpp;
kmy money2/kmy money2/knewbankdlg.cpp,v <-- knewbankdlg.cpp
new revision: 1.10; previous revision: 1.9
done
thb:~>
```

Note

If the option `-m` and the descriptive text is omitted on the command line, **CVS** starts an editor where the developer has to enter a descriptive text about the changes and save that file. Performing checkin operations that way is meaningful, if the description is longer or covers more than one file.

At this point, the changes are stored in the repository. An automatic mail is generated and send to the `kmymoney2-developer` mailing list `<kmymoney2-developer@lists.sourceforge.net>`. This mail informs all other developers about your changes and is an indication for them to update their sandboxes. The contents of the mail looks something like this:

```
From: Thomas Baumgart <ipwizard@users.sourceforge.net>
To: kmymoney2-developer@lists.sourceforge.net
Date: Sat, 24 Nov 2001 12:23:00 -0800
Subject: [Kmymoney2-developer] CVS update:

Update of /cvsroot/kmymoney2/kmymoney2/kmymoney2
In directory usw-pr-cvs1:/tmp/cvs-serv6662

Modified Files:
    knewbankdlg.cpp
Log Message:
Avoid return in the middle of a function

-----
Kmymoney2-developer mailing list
Kmymoney2-developer@lists.sourceforge.net
https://lists.sourceforge.net/lists/listinfo/kmymoney2-developer
```

While you checkin your changes, you should maintain the file **kmymoney2/ChangeLog**. You could probably use the same comments that you use for checkin in your changes or a more general note for many changes. That depends on your changes. Once all your changes are checked in, you also checkin **kmymoney2/ChangeLog**.

Updating changes performed by other developers

In case you noticed that other developers changed the repository - fortunately you will be noticed by a mail to the developer mailing list if that is the case - you should get those changes to your sandbox. This is accomplished using the **update** command of **CVS**.

Example A.4. Updating the sandbox

To update the local sandbox the following command is used. As most other **CVS** commands, it operates recursively from the current working directory in the sandbox.

```
thb:~> cvs -q upd
U README
M knewbankdlg.cpp
thb:~>
```

The above shown output has the following meaning: the file **README** is updated (U) from the repository to the sandbox because it has been changed by someone else in the meantime. The contents of the file in the sandbox will be replaced by the contents of the file in the repository, because it has

not been altered in the sandbox. The file **knewbankdlg.cpp** has been modified (M) in the sandbox and needs to be returned to the repository.

If you run the same command again, the output will change, as the file **README** is now up-to-date.

```
thb:~> cvs -q upd
M knewbankdlg.cpp
thb:~>
```

Sometimes it is useful to get an overview of what the status of certain files in the repository is without modifying the sandbox (updating). This can be accomplished by using the **-n** option to the update command.

Example A.5. Checking the status of files in the sandbox

```
thb:~> cvs -nq upd
U README
M knewbankdlg.cpp
thb:~>
```

The status of the files is the same as explained above, but the file **README** will *not* be updated. It remains unchanged in the sandbox. If you run this command again, the output remains.

Dismissing changes

It happens, that a developer tries to modify the source to gain a certain functionality and then wants to discard the changes. This is no problem at all with **CVS**. All the developer needs to do is to remove the file in the sandbox and run the **update** command of **CVS**. This will transfer the original version of the file in question to the sandbox.

Let's assume, that the changes made to **knewbankdlg.cpp** as outlined in a previous chapter should be dismissed. The following commands perform this operation:

Example A.6. Reverting changes made to the sandbox

```
thb:~> cvs -q upd
M knewbankdlg.cpp
thb:~> rm knewbankdlg.cpp
thb:~> cvs -q upd
U knewbankdlg.cpp
thb:~> cvs -q upd
thb:~>
```

Keeping different branches on the same machine

Whenever a configuration manager of the project decides to create a new stable release, the developers face a problem: they are not allowed to add new features to the software, only bug-fixes can be checked into the repository. Until the configuration manager opens the sources for further development, the developers are stuck.

To avoid this dilemma, the configuration manager creates a branch off the main development line when he creates the new stable release. Fixes will be made to the release-branch, new developments will be made to the main branch. This eliminates two problems: the configuration manager does not have to lock the current stage and the developers can continue with the implementation of features planned for the next release. Nevertheless, the stable version can be changed (fixes can be applied) and those fixes can be transferred to the main development branch so that they do not show up in future releases of the software.

Since in our project the developers will work on both, bug fixes and new development at the same time, it is convenient to have two sandboxes on the development machine. For the following examples, I have two subdirectories in my \$HOME for the project. One is for the release and the other for the development branch. I name them **stable** for the release branch and **devel** for the development branch.

The development branch

The development branch is the same as you use it today. Just move it from its current location to \$HOME/devel. I kept it directly in my \$HOME directory so I did it as follows:

Example A.7. Keeping stable and development branch on one machine

```
thb:~> md devel
thb:~> md stable
thb:~> mv kmymoney2 devel
thb:~>
```

Now the development sources are found in ~/devel/kmymoney2/. It is important to move all the CVS directories as well. If you start from scratch, then you just follow the instructions on how to checkout the project and do that in the **devel** subdirectory. See the chapter *Checking out from the repository* for an example.

The release branch

As soon as a release branch has been created by the configuration manager, you should get it to the stable directory. You do this by checking it out with the tag that has been assigned. The conventions have been defined in the chapter about Version management. For this example, I assume that a release branch for version 0.4 exists in the repository. ^[1]

Example A.8. Checking out the stable branch for the first time

```
thb:~> cd stable
thb:~/stable> cvs -d username@kmymoney2.cvs.sourceforge.net:/cvsroot/kmymoney2 \ [2]
    co -r rel-0-4-branch kmymoney2
thb:~/stable>
```

At this point it is important to use the *branch-tag* to be able to modify the files and check them back into the repository. If you are in the subdirectory containing the release-branch and you perform a **cvs update**, you will only get those changes, that were made on the branch. Also, changes are checked back into the release branch and do *NOT* show up on the development branch.

Note

If you want to keep more than one stable branch on your development machine, you can add the version number to the stable directory (e.g. stable-0.4, etc.)

Promoting bug-fixes to the main branch

Usually, changes made to the release-branch fix a problem. In many cases the problem still exists in the development branch. Therefore, it is necessary to promote the changes made on the release branch back to the development branch.

In most cases, it is very easy to promote the changes. The developer must be very careful though, as the fix might not be applicable in its form to the development branch anymore as things might have changed drastically due to new features. ^[3]

In this example, I assume changes were made to the single file README. ^[4] A complex fix could cover many files. The procedure described in the following is then required for each of them separately. Further on, I assume that the change has been checked in and that the revision was 1.14.2.1 before the fix and is now 1.14.2.2.

Example A.9. Promoting a change from the release to the development branch

```
thb:~> cd devel/kmymoney2
thb:~/devel/kmymoney2> cvs -q upd -j 1.14.2.1 -j 1.14.2.2 README
thb:~/devel/kmymoney2> vi README
thb:~/devel/kmymoney2> cvs ci -m "Included fix #493920" README
thb:~/devel/kmymoney2>
```

First, I go into the devel directory. Then I promote the changes to the README file in the development branch from the repository, verify the changes made (and possibly correct them) and checkin the changes to the development branch. That's it! ^[5]

Note

It's important to perform this procedure for every file affected by the fix separately, as the revision numbers usually differ significantly between the files. Also, I suggest to fix each problem separately. This reduces further problems while promoting the changes back to the development branch (e.g. one can leave out a fix completely if it does not apply at all to the development branch).

If the fix is very simple, it can certainly be promoted manually to the development directory tree by merely re-typing it.

Creating a new stable release

The procedure that needs to be followed is defined in a previous chapter. On the first glimpse, it seems rather complicated but it is not.

If you follow the procedure as it has been defined, you will understand the commands in our example. I assume to have a current unstable version of 0.3.x which will result in the stable version 0.4 and the next development version of 0.5.x. Further on, I assume, that I already differentiate between development and release directories. Also, the version number maintained with KDevelop is setup

correctly (0.4.pre1) and all files are checked into the repository.

Example A.10. Creating a new stable branch

```
thb:~> cd devel/kmymoney2
thb:~/devel/kmymoney2> cvs tag rel-0-4-pre1
thb:~/devel/kmymoney2> cvs tag -b rel-0-4-branch
```

Now modify the version number in KDevelop to 0.5.0, regenerate the files and checkin the changes as usual.

```
thb:~/devel/kmymoney2> cvs tag rel-0-5-base
thb:~/devel/kmymoney2>
```

Tip

Because I know, that I will need the branch sooner or later to fix some problems, I continue to check it out into the stable directory. See Keeping different branches on the same machine for details.

[1] Guess when I wrote this chapter ;-)

[2] The back-slash is inserted here to break the line for readability. For real usage, the command has to be entered on a single line.

[3] This is one of the reasons, why I suggest to apply the fix to the release branch and promote it to the developer branch, as the fix as it works on the release branch might break things and broken software is definitely something we do not want to happen on the stable branch.

[4] Fortunately, the error found was a documentation problem ;-)

[5] Of course, a fix to a source code file would be followed by a visual inspection (that's where **vi** or **kdevelop** come into play) and a compile run with further testing before the change is checked back into the repository.

Appendix B. Source and Header Examples

Table of Contents

Header File Example
Source File Example

This appendix contains an example of a header file listing and a source file listing.

Header File Example

```
/*
 *
 * ksettingsdlg.h
 *
 * -----
 * copyright      : (C) 2000,2001 by Michael Edwardes
 * email         : mte@users.sourceforge.net
 *
 */
```

```

*
*   This program is free software; you can redistribute it and/or modify
*   it under the terms of the GNU General Public License as published by
*   the Free Software Foundation; either version 2 of the License, or
*   (at your option) any later version.
*
*
*****/
#endif KSETTINGSDLG_H
#define KSETTINGSDLG_H

// -----
// QT Includes
#include <qcheckbox.h>
#include <qradiobutton.h>
#include <qbuttongroup.h>
#include <qcolor.h>
#include <qfont.h>

// -----
// KDE Includes
#include <kdialogbase.h>
#include <kfontdialog.h>
#include <kcolorbutton.h>
#include <klineedit.h>

// -----
// Project Includes

/**
 * This class is used to manipulate all the settings available for
 * KMyMoney2. It currently stores the values for the list settings
 * and whether to show the start dialog when KMyMoney2 starts.
 *
 * It uses KDialogBase to implement it's interface.
 *
 * It uses the global KConfig object to read and write the application
 * settings.
 *
 * @see KDialogBase
 *
 * @author Michael Edwardes 2000-2001
 * $Id: src-examples.docbook,v 1.3 2005/05/27 19:05:18 ipwizard Exp $
 *
 * @short A class to manipulate the settings needed for running KMyMoney2
 */
class KSettingsDlg : public KDialogBase {
    Q_OBJECT

private:
    /** Start prompt dialog */
    QRadioButton *m_qradiobuttonStartPrompt;
    /** Start file */
    QRadioButton *m_qradiobuttonStartFile;
    /** Color list */
    KColorButton *m_kcolorbuttonList;
    /** Color background */
    KColorButton *m_kcolorbuttonBack;
    /** Select font header */
    KFontChooser *m_kfontchooserHeader;
    /** Font cell setting */
    KFontChooser *m_kfontchooserCell;

```

```

/** No rows to show in register */
KLineEdit *m_klineeditRowCount;

/** Show grid in register ? */
QCheckBox *m_qcheckboxShowGrid;

QRadioButton *m_qradiobuttonPerTransaction;
QRadioButton *m_qradiobuttonOtherRow;

/** Set page general */
void setPageGeneral();
/** Set page list settings */
void setPageList();
/** Write settings */
void configWrite();
/** Read settings */
void configRead();

/** Attributes to store the variables so they can be undone when
 * cancelling the apply.
 */
QColor m_qcolorTempList;
QColor m_qcolorTempListBG;
QFont m_qfontTempHeader;
QFont m_qfontTempCell;
QString m_qstringTempRowCount;
bool m_bTempShowGrid;
bool m_bTempColourPerTransaction;
bool m_bTempStartPrompt;
bool m_bDoneApply;

private slots:
/** Called when OK pressed */
void slotOk();

/** Called when Apply pressed */
void slotApply();

/** Called when Cancel pressed */
void slotCancel();

/** Called when Reset pressed */
void slotUser1();

public:
/**
 * Standard constructor.
 *
 * @param parent The QWidget this is used in.
 * @param name The QT name.
 * @param modal True if we want the dialog to be application modal.
 *
 * @return An object of type KSettingsDlg.
 *
 * @see ~KSettingsDlg
 */
KSettingsDlg(QWidget *parent=0, const char *name=0, bool modal=true);

/**
 * Standard destructor.
 */

```

```

        * @return Nothing.
        *
        * @see KSettingsDlg
    **/
    ~KSettingsDlg();

signals:
    /**
     * Emitted when the Apply button is clicked to allow the application to
     * show the changes without having to close the dialog.
     *
     * @return Nothing
    **/
    void signalApply();
};

#endif

```

Source File Example

```

/*****
                                ksettingsdlg.cpp
                                -----
    copyright          : (C) 2000,2001 by Michael Edwardes
    email              : mte@users.sourceforge.net
    *****/

/*****
 *
 * This program is free software; you can redistribute it and/or modify
 *
 * it under the terms of the GNU General Public License as published by
 *
 * the Free Software Foundation; either version 2 of the License, or
 *
 * (at your option) any later version.
 *
 *
 *****/

// -----
// QT Includes
#include <qlayout.h>
#include <qvbox.h>
#include <qlabel.h>
#include <qgroupbox.h>
#include <qtabwidget.h>
#include <qvalidator.h>

// -----
// KDE Includes
#include <klocale.h>
#include <kstddirs.h>
#include <kiconloader.h>
#include <kconfig.h>
#include <kcolorbutton.h>
#include <kmessagebox.h>

// -----
// Project Includes
#include "ksettingsdlg.h"

```

```

/** Standard constructor for the class.
 * The constructor passes some additional parameters to the base class
 * KDialogBase
 * to set the buttons to be showed and the type of dialog to be shown.
**/
KSettingsDlg::KSettingsDlg(QWidget *parent, const char *name, bool modal)
: KDialogBase(IconList, i18n("Configure"), Ok|Cancel|Apply|User1, Ok, parent,
  name, modal, true, i18n("&Reset"))
{
  // Setup the pages and then read the configuration object.
  setPageGeneral();
  setPageList();
  configRead();
  m_bDoneApply=false;
}

/** Standard destructor for the class.
**/
KSettingsDlg::~KSettingsDlg()
{
}

/** Called to create the General page to be shown in the dialog.
**/
void KSettingsDlg::setPageGeneral()
{
  // Create the main frame to hold the widgets
  QVBox *qvboxMainFrame = addVBoxPage( i18n("General"), i18n("General settings"),
    DesktopIcon("misc"));

  // Create a group box to hold the available options
  QButtonGroup *qbuttongroup = new QButtonGroup(qvboxMainFrame, "GroupBox1");
  qbuttongroup->setTitle( i18n( "Startup options" ) );
  qbuttongroup->setColumnLayout(0, Qt::Vertical );
  qbuttongroup->layout()->setSpacing( 0 );
  qbuttongroup->layout()->setMargin( 0 );

  // Create a layout to organize the widgets.
  QVBoxLayout *qvboxlayout = new QVBoxLayout(qbuttongroup->layout());
  qvboxlayout->setAlignment( Qt::AlignTop );
  qvboxlayout->setSpacing( 6 );
  qvboxlayout->setMargin( 11 );

  // Create a check box to be in the group box
  m_qradiobuttonStartPrompt = new QRadioButton("start_prompt", qbuttongroup);
  m_qradiobuttonStartPrompt->setText( i18n( "Start with dialog prompt (default)" ) );
  qvboxlayout->addWidget(m_qradiobuttonStartPrompt);

  // Create another check box to the group box
  m_qradiobuttonStartFile = new QRadioButton("start_file", qbuttongroup);
  m_qradiobuttonStartFile->setText( i18n( "Start with last file used" ) );
  qvboxlayout->addWidget(m_qradiobuttonStartFile);
}

/** Called to create the Main List page shown in the dialog.
**/
void KSettingsDlg::setPageList()
{
  // Create the page.
  QVBox *qvboxMainFrame = addVBoxPage( i18n("Main List"), i18n("List settings"),
    locate("appdata", "pics/setting_list.png"));
}

```

```

// Create the tab widget
QTabWidget *qtabwidget = new QTabWidget(qvboxMainFrame, "TabWidget2");

// Create the first page
QWidget *qwidgetPage = new QWidget(this);

// Create the layout for the page
QVBoxLayout *qvboxlayoutPage = new QVBoxLayout(qwidgetPage);
qvboxlayoutPage->setSpacing( 6 );
qvboxlayoutPage->setMargin( 11 );

// Create a horizontal layout to hold two widgets
QHBoxLayout *qhboxlayout = new QHBoxLayout;
qhboxlayout->setSpacing( 6 );
qhboxlayout->setMargin( 0 );

// Create the first widget
QLabel *qlabel = new QLabel(i18n("Number of lines in the register view:"),
    qwidgetPage);
qhboxlayout->addWidget(qlabel);

// Create the second widget
m_klineeditRowCount = new KLineEdit(qwidgetPage);
QIntValidator *qintvalidator = new QIntValidator(1, 3, m_klineeditRowCount);
m_klineeditRowCount->setValidator(qintvalidator);
qhboxlayout->addWidget(m_klineeditRowCount);

// Add the horizontal layout
qvboxlayoutPage->addLayout(qhboxlayout);

// Ceate another widget
m_qcheckboxShowGrid = new QCheckBox(i18n("Show a grid in the register view"),
    qwidgetPage);
qvboxlayoutPage->addWidget(m_qcheckboxShowGrid);

// Create a group to hold two radio buttons
QButtonGroup *qbuttongroup = new QButtonGroup(qwidgetPage, "ButtonGroup1");
qbuttongroup->setTitle(i18n("Row Colour options"));
qbuttongroup->setColumnLayout(0, Qt::Vertical );
qbuttongroup->layout()->setSpacing( 0 );
qbuttongroup->layout()->setMargin( 0 );

// Create a layout
QVBoxLayout *qvboxlayout = new QVBoxLayout(qbuttongroup->layout());
qvboxlayout->setAlignment( Qt::AlignTop );
qvboxlayout->setSpacing( 6 );
qvboxlayout->setMargin( 11 );

// Add the first radio button
m_qradiobuttonPerTransaction = new QRadioButton(qbuttongroup, "m_per_trans");
m_qradiobuttonPerTransaction->setText( i18n("Use one colour per transaction" ) );
qvboxlayout->addWidget(m_qradiobuttonPerTransaction);

// Add the second radio button
m_qradiobuttonOtherRow = new QRadioButton(qbuttongroup, "m_every_other");
m_qradiobuttonOtherRow->setText( i18n( "Change colour every other row" ) );
qvboxlayout->addWidget(m_qradiobuttonOtherRow);
qvboxlayoutPage->addWidget(qbuttongroup);

// Add a vertical spacer to take up the remaining available space
QSpacerItem* spacer = new QSpacerItem( 20, 20, QSizePolicy::Minimum,

```

```

    QSizePolicy::Expanding );
qvbboxlayoutPage->addItem( spacer );

// Add the page to the tab
qtabwidget->insertTab(qwidgetPage, i18n("General"));

// Create a new tab for the colour options
QWidget *qwidgetColour = new QWidget(qtabwidget, "tab");

// Create a vertical layout
QVBoxLayout *qvbboxlayoutColour = new QVBoxLayout(qwidgetColour);
qvbboxlayoutColour->setSpacing( 6 );
qvbboxlayoutColour->setMargin( 11 );

// Create a horizontal layout to include the label and button
QHBoxLayout *qhboxlayoutColour = new QHBoxLayout;
qhboxlayoutColour->setSpacing( 6 );
qhboxlayoutColour->setMargin( 0 );

// Add the label and button
QLabel *qlabelListColour = new QLabel(i18n( "List view colour :" ),
    qwidgetColour);
qhboxlayoutColour->addWidget(qlabelListColour);
m_kcolorbuttonList = new KColorButton(qwidgetColour, "colour_list");
qhboxlayoutColour->addWidget(m_kcolorbuttonList);

// Add the horizontal layout
qvbboxlayoutColour->addLayout(qhboxlayoutColour);

// Create another horizontal layout to include the label and button
QHBoxLayout *qhboxlayoutBGColour = new QHBoxLayout;
qhboxlayoutBGColour->setSpacing( 6 );
qhboxlayoutBGColour->setMargin( 0 );

// Add the label and button
QLabel *qlabelListBGColour = new QLabel(i18n( "List background colour :" ),
    qwidgetColour);
qhboxlayoutBGColour->addWidget(qlabelListBGColour);
m_kcolorbuttonBack = new KColorButton(qwidgetColour, "colour_back");
qhboxlayoutBGColour->addWidget(m_kcolorbuttonBack);

// Add the horizontal layout
qvbboxlayoutColour->addLayout(qhboxlayoutBGColour);

// Add a vertical spacer to take up the remaining available space
QSpacerItem* qspaceritemColour = new QSpacerItem( 20, 20,
    QSizePolicy::Minimum, QSizePolicy::Expanding );
qvbboxlayoutColour->addItem( qspaceritemColour );

// Add the page to the tab widget
qtabwidget->insertTab(qwidgetColour, i18n( "Color" ));

// Create another tab adding a font chooser widget
QVBoxLayout *qvbboxInsideTab1 = new QVBoxLayout( this, "tab1" );
qvbboxInsideTab1->setSpacing( 6 );
qvbboxInsideTab1->setMargin( 11 );
m_kfontchooserHeader = new KFontChooser(qvbboxInsideTab1);
qtabwidget->insertTab(qvbboxInsideTab1, i18n("Header Font"));

// Create another tab adding a font chooser widget
QVBoxLayout *qvbboxInsideTab2 = new QVBoxLayout( this, "tab2" );
qvbboxInsideTab2->setSpacing( 6 );

```

```

    qvboxInsideTab2->setMargin( 11 );
    m_kfontchooserCell = new KFontChooser(qvboxInsideTab2);
    qtabwidget->addTab(qvboxInsideTab2, i18n("Cell Font"));
}

/** Read all the settings in from the global KConfig object and set all the
 * widgets appropriately.
 */
void KSettingsDlg::configRead()
{
    KConfig *kconfig = KGlobal::config();
    kconfig->setGroup("Settings Dialog");
    QSize *qsizeDefaultSize = new QSize(470,470);
    this->resize(kconfig->readSizeEntry("Geometry", qsizeDefaultSize));

    kconfig->setGroup("General Options");
    m_bTempStartPrompt = kconfig->readBoolEntry("StartDialog", true);
    m_qradiobuttonStartPrompt->setChecked(m_bTempStartPrompt);
    m_qradiobuttonStartFile->setChecked(!m_bTempStartPrompt);

    kconfig->setGroup("List Options");

    QFont qfontDefault = QFont("helvetica", 12);
    QColor qcolorDefault = Qt::white;
    QColor qcolorDefaultBG = Qt::gray;

    m_qcolorTempList = kconfig->readColorEntry("listColor", &qcolorDefault);
    m_kcolorbuttonList->setColor(m_qcolorTempList);

    m_qcolorTempListBG = kconfig->readColorEntry("listBGColor",
        &qcolorDefaultBG);
    m_kcolorbuttonBack->setColor(m_qcolorTempListBG);

    m_qfontTempHeader = kconfig->readFontEntry("listHeaderFont",
        &qfontDefault);
    m_kfontchooserHeader->setFont(m_qfontTempHeader);

    m_qfontTempCell = kconfig->readFontEntry("listCellFont", &qfontDefault);
    m_kfontchooserCell->setFont(m_qfontTempCell);

    m_qstringTempRowCount = kconfig->readEntry("RowCount", "2");
    m_klineditRowCount->setText(m_qstringTempRowCount);

    m_bTempShowGrid = kconfig->readBoolEntry("ShowGrid", true);
    m_qcheckboxShowGrid->setChecked(m_bTempShowGrid);

    m_bTempColourPerTransaction =
        kconfig->readBoolEntry("ColourPerTransaction", true);
    m_qradiobuttonPerTransaction->setChecked(m_bTempColourPerTransaction);
    m_qradiobuttonOtherRow->setChecked(!m_bTempColourPerTransaction);
}

/** Write out all the settings to the global KConfig object.
 */
void KSettingsDlg::configWrite()
{
    KConfig *kconfig = KGlobal::config();
    kconfig->setGroup("Settings Dialog");
    kconfig->writeEntry("Geometry", this->size() );

    kconfig->setGroup("List Options");
    kconfig->writeEntry("listColor", m_kcolorbuttonList->color());
}

```

```

kconfig->writeEntry("listBGColor", m_kcolorbuttonBack->color());
kconfig->writeEntry("listHeaderFont", m_kfontchooserHeader->font());
kconfig->writeEntry("listCellFont", m_kfontchooserCell->font());
kconfig->writeEntry("RowCount", m_klineeditRowCount->text());
kconfig->writeEntry("ShowGrid", m_qcheckboxShowGrid->isChecked());
kconfig->writeEntry("ColourPerTransaction",
    m_qradiobuttonPerTransaction->isChecked());

kconfig->setGroup("General Options");
kconfig->writeEntry("StartDialog",
    m_qradiobuttonStartPrompt->isChecked());

kconfig->sync();
}

/** Called on OK being pressed */
void KSettingsDlg::slotOk()
{
    int nCount = m_klineeditRowCount->text().toInt();
    if (nCount <= 0 || nCount >= 4) {
        KMessageBox::information(this, i18n("The row count has to be between 1
and 3"));
        m_klineeditRowCount->setFocus();
        return;
    }
    configWrite();
    this->accept();
}

/** Called on Apply being pressed */
void KSettingsDlg::slotApply()
{
    int nCount = m_klineeditRowCount->text().toInt();
    if (nCount <= 0 || nCount >= 4) {
        KMessageBox::information(this, i18n("The row count has to be between 1
and 3"));
        m_klineeditRowCount->setFocus();
        return;
    }
    m_bDoneApply = true;
    configWrite();
    emit signalApply();
}

/** Called on Cancel being pressed.
 * It writes out all the original settings read when it was created.
 */
void KSettingsDlg::slotCancel()
{
    // make sure the config object is the same as we left it
    KConfig *kconfig = KGlobal::config();
    kconfig->setGroup("List Options");
    kconfig->writeEntry("listColor", m_qcolorTempList);
    kconfig->writeEntry("listBGColor", m_qcolorTempListBG);
    kconfig->writeEntry("listHeaderFont", m_qfontTempHeader);
    kconfig->writeEntry("listCellFont", m_qfontTempCell);
    kconfig->writeEntry("RowCount", m_qstringTempRowCount);
    kconfig->writeEntry("ShowGrid", m_bTempShowGrid);
    kconfig->writeEntry("ColourPerTransaction", m_bTempColourPerTransaction);

    kconfig->setGroup("General Options");
    kconfig->writeEntry("StartDialog", m_bTempStartPrompt);

```

```

kconfig->sync();

if (m_bDoneApply)
    accept();
else
    reject();
}

/** Called when the user presses the User1 button. In our case that is the
 * reset button.
 *
 * It just resets all the attributes to the values read on creation.
 */
void KSettingsDlg::slotUser1()
{
    m_qradiobuttonStartPrompt->setChecked(m_bTempStartPrompt);
    m_kcolorbuttonList->setColor(m_qcolorTempList);
    m_kcolorbuttonBack->setColor(m_qcolorTempListBG);
    m_kfontchooserHeader->setFont(m_qfontTempHeader);
    m_kfontchooserCell->setFont(m_qfontTempCell);
    m_klineditRowCount->setText(m_qstringTempRowCount);
    m_qcheckboxShowGrid->setChecked(m_bTempShowGrid);
    m_qradiobuttonPerTransaction->setChecked(m_bTempColourPerTransaction);
    m_qradiobuttonOtherRow->setChecked(!m_bTempColourPerTransaction);
}

```

Appendix C. Unit Test Examples

Table of Contents

Unit Test Header File Example

Unit Test Source File Example

Unit Test Container Source File Example

This appendix contains an example of a unit test header file listing and a unit test source file listing as well as a unit test container source file listing.

Unit Test Header File Example

```

/*****
                                mymoneyexceptiontest.h
                                -----
copyright                       : (C) 2002 by Thomas Baumgart
email                           : ipwizard@users.sourceforge.net
*****/

/*****
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 *****/

#ifdef __MYMONEYEXCEPTIONTEST_H__
#define __MYMONEYEXCEPTIONTEST_H__

```

```

#include <cppunit/extensions/HelperMacros.h>

#define private public
#include "mymoneyutils.h"
#include "mymoneyexception.h"
#undef private

class MyMoneyExceptionTest : public CppUnit::TestFixture {
    CPPUNIT_TEST_SUITE(MyMoneyExceptionTest);
    CPPUNIT_TEST(testDefaultConstructor);
    CPPUNIT_TEST(testConstructor);
    CPPUNIT_TEST_SUITE_END();

protected:
public:
    MyMoneyExceptionTest();

    void setUp();

    void tearDown();

    void testDefaultConstructor();

    void testConstructor();

};
#endif

```

Unit Test Source File Example

```

/*****
                                mymoneyexceptiontest.cpp
                                -----
copyright      : (C) 2002 by Thomas Baumgart
email         : ipwizard@users.sourceforge.net
*****/

/*****
*
*   This program is free software; you can redistribute it and/or modify
*   it under the terms of the GNU General Public License as published by
*   the Free Software Foundation; either version 2 of the License, or
*   (at your option) any later version.
*
*****/

#include "mymoneyexceptiontest.h"

MyMoneyExceptionTest::MyMoneyExceptionTest()
{
}

void MyMoneyExceptionTest::setUp()
{
}

void MyMoneyExceptionTest::tearDown()
{
}

```

```

}

void MyMoneyExceptionTest::testDefaultConstructor()
{
    MyMoneyException *e = new MYMONEYEXCEPTION("Message");
    CPPUNIT_ASSERT(e->what() == "Message");
    CPPUNIT_ASSERT(e->line() == __LINE__-2);
    CPPUNIT_ASSERT(e->file() == __FILE__);
    delete e;
}

void MyMoneyExceptionTest::testConstructor()
{
    MyMoneyException *e = new MyMoneyException("New message",
                                                "Joe's file", 1234);
    CPPUNIT_ASSERT(e->what() == "New message");
    CPPUNIT_ASSERT(e->line() == 1234);
    CPPUNIT_ASSERT(e->file() == "Joe's file");
    delete e;
}

```

Unit Test Container Source File Example

This test environment also contains a reference to a memory usage checker which can safely be ignored. It is also contained in the **KMyMoney** environment and is a great help if looking for memory leaks. Also notice the usage of the C++ preprocessor directive *#ifndef HAVE_LIBCPPUNIT* to avoid compile errors when CPPUNIT is not installed.

Another specialty that is not required by CPPUNIT is the specific `TestProgressListener`. It is used here to print the name of the fixture that is currently running. Since this method is called upon start of each test case, some logic is necessary to print the name only once.

```

/*****
                                autotest.cpp
                                -----
copyright           : (C) 2002 by Thomas Baumgart
email               : ipwizard@users.sourceforge.net
*****/

/*****
*
* This program is free software; you can redistribute it and/or modify
* it under the terms of the GNU General Public License as published by
* the Free Software Foundation; either version 2 of the License, or
* (at your option) any later version.
*
*****/

#include "config.h"

#include <iostream>

#ifdef HAVE_LIBCPPUNIT

#include "cppunit/TextTestRunner.h"
#include "cppunit/TextTestResult.h"
#include "cppunit/TestSuite.h"
#include "cppunit/extensions/HelperMacros.h"

```

```

#include "mymoneyexceptiontest.h"

#include "cppunit/TextTestProgressListener.h"

class MyProgressListener : public CppUnit::TextTestProgressListener
{
    void startTest(CppUnit::Test *test) {
        QString name = test->getName().c_str();
        name = name.mid(2); // cut off first 2 chars
        name = name.left(name.find('.'));
        if(m_name != name) {
            if(m_name != "")
                cout << endl;
            cout << "Running: " << name << endl;
            m_name = name;
        }
    }
private:
    QString m_name;
};

#endif

int
main(int argc, char** argv)
{
#ifdef HAVE_LIBCPPUNIT
#ifdef _CHECK_MEMORY
    _CheckMemory_Init(0);
#endif

    CPPUNIT_TEST_SUITE_REGISTRATION(MyMoneyExceptionTest);

    CppUnit::TestFactoryRegistry &registry =
        CppUnit::TestFactoryRegistry::getRegistry();
    CppUnit::Test *suite = registry.makeTest();

    CppUnit::TextTestRunner* runner = new CppUnit::TextTestRunner();
    runner->addTest(suite);

    MyProgressListener progress;
    runner->eventManager().addListener(&progress);
    runner->run();

    delete runner;

#ifdef _CHECK_MEMORY
    chkmem.CheckMemoryLeak( true );
    _CheckMemory_End();
#endif // _CHECK_MEMORY

#else
    std::cout << "libcppunit not installed. no automatic tests available."
              << std::endl;
#endif // HAVE_LIBCPPUNIT
    return 0;
}

```

Appendix D. RPM SPEC file example

This appendix contains an example of an RPM SPEC file.

```
#
# spec file for package kmymoney
#
# Copyright (c) 2002,2003,2004,2005 Thomas Baumgart
# This file and all modifications and additions to the pristine
# package are under the same license as the package itself.
#
# please send bugfixes or comments to kmymoney2-developer@lists.sourceforge.net
#

%define is_mandrake %{test -e /etc/mandrake-release && echo 1 || echo 0}
%define is_suse %{test -e /etc/SuSE-release && echo 1 || echo 0}
%define is_fedora %{test -e /etc/fedora-release && echo 1 || echo 0}

%define dist redhat
%define disttag rh

%if %is_mandrake
%define dist mandrake
%define disttag mdk
%endif
%if %is_suse
%define dist suse
%define disttag suse
%define kde_path /opt/kde3
%endif
%if %is_fedora
%define dist fedora
%define disttag rhfc
%endif

%define _bindir      %kde_path/bin
%define _datadir     %kde_path/share
%define _iconsdir   %_datadir/icons
%define _docdir     %_datadir/doc
%define _localedir  %_datadir/locale
%define qt_path      /usr/lib/qt3

%define distver %{release}"rpm -q --queryformat='%{VERSION}' %{{dist}}-release 2> /dev/null | tr . : | sed s://g'' ; if test $? != 0 ; then release="" ; fi ; echo "$release"}
%define distlibsuffix %{{_bindir}/kde-config --libsuffix 2> /dev/null}
%define _lib lib%distlibsuffix
%define packer %(finger -lp 'echo "$USER"' | head -n 1 | cut -d: -f 3)

Name:          kmymoney
Icon:          kmymoney.xpm
Summary:       The Personal Finances Manager for KDE.
Version:       0.8
Release:       1.{{disttag}}%{distver}
License:       GPL
Vendor:        The KMyMoney development team <kmymoney2-developers@lists.sourceforge.net>
Packager:      %packer
Group:         Productivity/Office/Finance
Source0:       %{name}2-{{version}}.tar.bz2
BuildRoot:     %{_tmpdir}/%{name}2-{{version}}-{{release}}-build
BuildRequires: kdesbase3-devel
Prereq:        /sbin/ldconfig

%description
KMyMoney is the Personal Finance Manager for the KDE environment.
It provides the functions required to balance your checkbooks,
manage your personal accounts, investments, loans and
categorise your incomes and expenses.

For the most up-to-date information and sources please
visit the project web-site at http://kmymoney2.sourceforge.net/.

To stay informed about new releases and other user related topics,
please register with the KMyMoney User Mailinglist. It's a low volume
mailing list. More information how to register can be found on the
project's web-site.

%package devel
#Requires:
Summary: KMyMoney development files
Group: Productivity/Office/Finance
Provides: kmymoney-devel

%description devel
This package contains necessary header files for KMyMoney development.

This package is necessary to compile plugins for KMyMoney.

%package ofx
Requires: kmymoney
Summary: KMyMoney OFX plugin
Group: Productivity/Office/Finance
Provides: kmymoney-ofx

%description ofx
This package contains necessary files for the KMyMoney OFX plugin.

%prep
#echo %_target
#echo %_target_alias
#echo %_target_cpu
#echo %_target_os
#echo %_target_vendor
echo Building %{name}-{{version}}-{{release}}

%setup -q -n %{name}2-{{version}}

%build
CFLAGS="%optflags* CXXFLAGS=%{optflags}* \
./configure --mandir=%{_mandir} \
--disable-rpath \
--with-xinerama \
--without-glib \
--disable-debug \
--disable-cppunit \
--enable-final"

make

%install
make DESTDIR=%buildroot install

%clean
```

```

[ ${RPM_BUILD_ROOT} != "/" ] && rm -rf ${RPM_BUILD_ROOT}

%post
cd %_docdir/HTML/*/%{name}2
ln -s ../common common
/sbin/ldconfig

%postun
/sbin/ldconfig

%files
%defattr(-,root,root)

%dir %_docdir/HTML/en/%{name}2/
%doc %_docdir/HTML/*/%{name}2/*.docbook
%doc %_docdir/HTML/*/%{name}2/*.png
%doc %_docdir/HTML/*/%{name}2/index.cache.bz2

# the binary files
%{_bindir}/%{name}
%{_bindir}/%{name}2

# the shared libraries
%kde_path/%_lib/*.so.*.*

#
%dir %_datadir/apps/
%dir %_datadir/apps/%{name}2/
%dir %_datadir/apps/%{name}2/html
%dir %_datadir/apps/%{name}2/templates
%dir %_datadir/apps/%{name}2/templates/C
%dir %_datadir/apps/%{name}2/templates/de_DE
%dir %_datadir/apps/%{name}2/templates/en_GB
%dir %_datadir/apps/%{name}2/templates/en_US
%dir %_datadir/apps/%{name}2/templates/fr_FR
%dir %_datadir/apps/%{name}2/templates/pt_BR
%dir %_datadir/apps/%{name}2/templates/ru_SU
%_datadir/apps/%{name}2/templates/README
%_datadir/apps/%{name}2/templates/*/*.*.kmt

%_datadir/apps/%{name}2/*rc
%_datadir/apps/%{name}2/html/*
%_datadir/apps/%{name}2/tips

%dir %_datadir/apps/%{name}2/pics/
%_datadir/apps/%{name}2/pics/*.*.png

%dir %_datadir/apps/%{name}2/icons/
%dir %_datadir/apps/%{name}2/icons/hicolor/
%dir %_datadir/apps/%{name}2/icons/hicolor/16x16/
%dir %_datadir/apps/%{name}2/icons/hicolor/16x16/actions/
%dir %_datadir/apps/%{name}2/icons/hicolor/22x22/
%dir %_datadir/apps/%{name}2/icons/hicolor/22x22/actions/
%dir %_datadir/apps/%{name}2/icons/hicolor/32x32/
%dir %_datadir/apps/%{name}2/icons/hicolor/32x32/apps
%dir %_datadir/apps/%{name}2/icons/hicolor/48x48/
%dir %_datadir/apps/%{name}2/icons/hicolor/48x48/apps
%dir %_datadir/apps/%{name}2/icons/hicolor/64x64/
%dir %_datadir/apps/%{name}2/icons/hicolor/64x64/apps
%_datadir/apps/%{name}2/icons/hicolor/*/*/*.*.png

#
#
%_datadir/applications/kde/kmymoney2.desktop
%_datadir/mimelnk/application/x-kmymoney2.desktop
%_datadir/servicetypes/*

#
#
%_iconsdir/*/*/*.*.png

#
#
%doc %_mandir/man1/kmymoney2.1.gz

#
#
%_localedir/*/*/*.*.mo

# AgBanking plugin related files
%dir %_datadir/apps/kmm_kbanking/
%_datadir/apps/kmm_kbanking/*rc

# plugin related files
%kde_path/%_lib/kde3/*.*.so

%files ofx
%_datadir/services/kmm_ofximport.desktop
%kde_path/%_lib/kde3/kmm_ofximport.so

%files devel
%kde_path/include/kmymoney/*
%kde_path/%_lib/*.*.la

# plugin related files
%kde_path/%_lib/kde3/*.*.la

%changelog
* Mon May 26 2005 - ipwizd (at) users.sourceforge.net
- Added kmymoney-ofx package

* Tue Mar 22 2005 - ipwizd (at) users.sourceforge.net
- Added more template functionality to provide more
distributions
- Added kmymoney-devel package

* Mon Jan 30 2005 - ipwizd (at) users.sourceforge.net
- Started adding distro independant layout

* Sat Oct 30 2004 - ipwizd (at) users.sourceforge.net
* Preparations for 0.6.3

* Thu Sep 23 2004 - ipwizd (at) users.sourceforge.net
* Preparations for 0.6.2

* Sat Jun 5 2004 - ipwizd (at) users.sourceforge.net
- Preparations for 0.6

* Thu Apr 22 2004 - ipwizd (at) users.sourceforge.net
- Preparations for 0.6rc4

* Fri Feb 20 2004 - ipwizd (at) users.sourceforge.net
- Removed the standard directories

```

- Uninstall the default account files also
- Preparations for 0.6rc3
- * Thu Feb 5 2004 - ipwizard (at) users.sourceforge.net
- Remove CVS directories from SRPMS
- Preparations for 0.6rc2
- * Mon Dec 29 2003 - ipwizard (at) users.sourceforge.net
- Preparations for 0.6rc1
- Incorporated some changes from SuSE distro
- added man page file
- * Thu Jan 9 2003 - ipwizard (at) users.sourceforge.net
- Added missing files home.html and kmymoney2.css
- * Mon Dec 16 2002 - ipwizard (at) users.sourceforge.net
- Removed make command only required for CVS download
- Update version to match filename
- * Sun Dec 15 2002 - ipwizard (at) users.sourceforge.net
- Updated for version 0.51
- * Tue Jan 15 2002 - ipwizard (at) users.sourceforge.net
- Initial implementation

Appendix E. Licence

Table of Contents

Free Documentation Licence

This document is released under Free Documentation licence; the terms of this licence are detailed below.

Free Documentation Licence

GNU Free Documentation License
Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either

commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions

(which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already

includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these

copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
A copy of the license is included in the section entitled "GNU
```

Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.