Effectively testing Qt 5 using *Squish Coco*

Sébastien Fricker and Amanda Burma

09/18/2012

1 Introduction

With more than tree million lines of code, Qt 5 is one of the largest open source projects. Creating a challenge for every contributing developer. In fact how can we be sure:

- code modifications are sufficiently tested?
- code modifications do not impact other features?
- a reviewer can approve a patch by simply reading a basic diff?

The approach to expect unit testing by a developer and letting the maintainer or approver take responsibility when integrating an extension, can be vastly improved by *Squish Coco* as it:

1. helps the Qt contributor find areas containing untested portions of code.
2. helps the approver verify the modification is thoroughly tested and no regressions exist.

2 System Prerequisites

2.1 Installing *Squish Coco*

Squish Coco is available in two editions:

- A Non-Commercial Edition which includes a compiler replacement, performing the code coverage analysis and a set of command line tools creating the ability to generate reports and manipulate code coverage data.
- A Professional Edition which includes also a graphical user interface, professional support and a license expanding its usage beyond non-commercial applications.

To install first sign up for an evaluation or permanent license:

for *Squish Coco*: `http://www.froglogic.com/squish/coco/`

for *Squish Coco Non-Commercial*: `http://www.froglogic.com/squish/coco/non-commercial.php`

This article only uses *Squish Coco*'s command line tools, allowing either edition to be used.
After installing *Squish Coco* modify the path variable to access `csg++`, the *Squish Coco* compiler replacement, from the command line.

2.2 System Prerequisites for Building Qt 5

Qt 5 depends of an important set of libraries. A full description of the requirements can be found on Qt homepage: `http://qt-project.org/wiki/Building_Qt_5_from_Git`

2.3 Downloading Qt 5 Source

The system variable `$WORK` refers to the working directory. `$QT5` is the Qt 5 source check out location. And `$PROCESSORS` indicates the number of available CPU cores.

```
export WORK=$HOME/coco  # or use an other location
export QT5=$WORK/qt5
export QT5BASE=$QT5/qtbase
export PROCESSORS=$(cat /proc/cpuinfo | grep processor | wc -l)

mkdir -p $WORK
cd $WORK
git clone git://gitorious.org/qt/qt5.git
cd $QT5
./init-repository -f
git pull
git submodule update
```

The section (above) downloads the complete Qt 5 source.

3 Building Qt 5 and Executing Unit Test

Qt 5 provides a command line switch called `-testcocoon`, which permits the instrumentation of *Squish Coco*̈More precisely:

- the Makefile uses `csg++` instead of `g++` for code generation.
- for each unit test execution, an execution report is generated, displaying the code coverage for one test or an entire test suite.

Execute the following to compile:

```
./configure -no-qpa-platform-guard -platform  linux-g++-64 \
            -developer-build -no-gtkstyle -no-pch \
            -opensource -testcocoon -confirm-license -prefix $QT5BASE
cd $QT5BASE
make -j$PROCESSORS
```

At this stage, QtBase is generated along with `$QT5BASE/lib` for each generated shared library a file with the extension `.csmes` is produced. This file contains the code coverage instrumentation, and upon completion also contains the code coverage report for each unit test.
Perform the following to execute the test suite: [1]

[1] -k is here necessary to avoid that make stops at the first unit test which fails.

```
make check -k
```

Now, as each test executes, the instrumentation files in `$QT5BASE/lib` are not updated, however import the unit test results.

4 Code Coverage Report of Qt 5 Libraries

The following suppresses all coverage reports from the `uic`, `moc` and `rcc` tools:

```
find $QT5 -name "rcc.csexe" -exec rm -v {} \;
find $QT5 -name "moc.csexe" -exec rm -v {} \;
find $QT5 -name "uic.csexe" -exec rm -v {} \;
```

The following imports the execution report (`.csexe` file) for each unit test into the instrumentation database (`.csmes` file):

```
cd $QT5BASE
find . -name "tst_*.csexe" | while read LINE
do
    CSEXE="$LINE"
    CSMES=$(dirname "$CSEXE")/$(basename "$CSEXE" .csexe).csmes
    if [ -e "$CSMES" ]
    then
        echo -n "Generating␣Unit␣Test␣Database␣$CSMES␣...␣"
        cmcsexeimport -m "$CSMES" -e "$CSEXE" -t "unittest" -p merge
        echo "done"
    fi
    rm "$CSEXE"
done
```

cmcsexeimport performs the import operation. After which the imported execution report is no longer needed, and can be safely removed.
At this stage an instrumentation database exists for each unit test, which contains the code coverage information for each execution. The following retrieves and places this information into the Qt 5 libraries' instrumentation databases:

```
cd $QT5BASE
for QTLIB in lib/lib*.csmes
do
    QTLIBTMP="$1".tmp
    UNITTESTS=$(find . -name "tst_*.csmes")
    echo -n "Importing␣Unit␣Tests␣Result␣in␣$QTLIB␣...␣"
    cmmerge -o "$QTLIBTMP"  -i "$QTLIB" $UNITTESTS
    mv -f "$QTLIBTMP" "$QTLIB"
    echo "done"
done
```

cmmerge imports the unit test results into Qt 5 library's database. It also ignores the code coverage in the test code. After this operation, the instrumentation database for each unit test is no longer needed.
The following generates an HTML report from the code coverage database:

```
cd $QT5BASE
for QTLIB in lib/lib*.csmes
do
    HTML=$(dirname "$QTLIB")/$(basename "$QTLIB" .csmes).html
    TTTLE=$(basename "$QTLIB" .csmes)
    cmreport -m "$QTLIB" -s '.*' --html="$HTML" --toc --title="$TITLE" \
             --source=all --method=all --global=all --dead-code \
             -bargraph --source-sort=coverage --method-sort=coverage
done
```

In `$QT5BASE/lib` an HTML report is generated for each Qt library, making it possible to analyze which source code line was not covered by the test suite.

5 Using *Squish Coco* Qt 5 Development

Having a code coverage report of Qt 5 in its entirety does not have any real value for a Qt developer: The developer is only focused on developing a feature or fixing a bug and does not need to be aware of the quality of the entire Qt project. Instead, the developer needs to know how the modification can negatively impact the project and how well the changes are tested.

5.1 Hacking in Qt 5

The following example demonstrates how *Squish Coco* can be used to better understand the impact of a change and the required testing: To support the € symbol when converting a Unicode string to a Latin-1 codec, the € (unicode 0x20AC) will be converted to the character 'E'.

Modify `QLatin1Codec::convertFromUnicode` as follows (modifications are underlined):

```
1  QByteArray QLatin1Codec::convertFromUnicode(const QChar *ch,
2                                              int len, ConverterState *state) const
3  {
4      const char replacement = (state && state->flags & ConvertInvalidToNull) ? 0 : '?';
5      QByteArray r(len, Qt::Uninitialized);
6      char *d = r.data();
7      int invalid = 0;
8      for (int i = 0; i < len; ++i) {
9          if (ch[i] == 0x20AC) {      // MODIFICATION: Euro symbol handling
10             d[i] = 'E';             // MODIFICATION: Euro symbol handling
11         } else if (ch[i] > 0xff) {
12             d[i] = replacement;
13             ++invalid;
14         } else {
15             d[i] = (char)ch[i].cell();
16         }
17     }
18     if (state) {
19         state->invalidChars += invalid;
20     }
21     return r;
22 }
```

Before generating the Qt library, make a copy of the current instrumentation database as follows:

```
cd $QT5BASE
export QT5REF=$QT5/../qt5_lib_ref/
mkdir -p $QT5REF
cp $QT5BASE/lib/*.csmes $QT5REF/
```

Then run make to compile. `$QT5BASE/lib/libQtCore.so.5.0.0.csmes` is updated at the same time as `$QT5BASE/lib/libQtCore.so.5.0.0.so`.
No tests are executed at this stage; however *Squish Coco* can analyze which test is impacted by the modification:

```
cd $QT5BASE
cmreport -m "$QT5REF/libQtCore.so.5.0.0.csmes" -s '.*' --html="euro_sym.html" \
         --toc --title="Euro␣symbol␣support" \
         --source=all --method=all --global=all \
         -bargraph --source-sort=coverage --method-sort=coverage \
         -mr "$QT5BASE/lib/libQtCore.so.5.0.0.csmes" --execution=all
```

`euro_sym.html` report is the coverage report for the modified functions, and in our case only the `QLatin1Codec::convertFromUnicode` function. The list of executions is limited to two entries:

1. `tst_QString` with 78% coverage
2. `tst_QTextStream` with 64% coverage

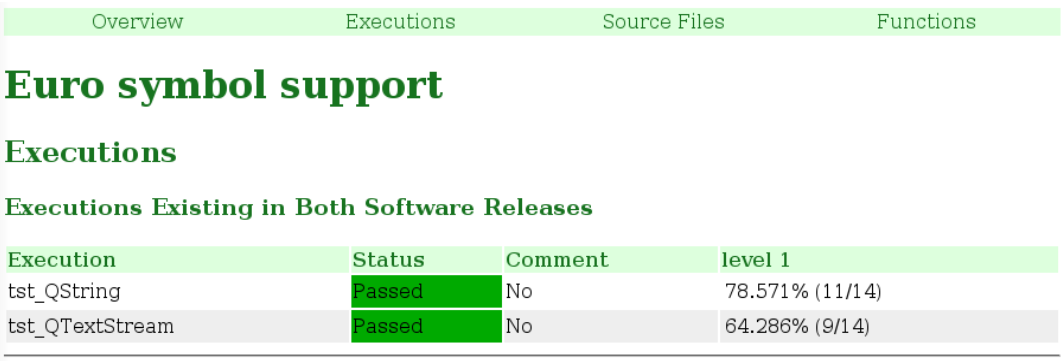


Figure 1: *Tests Impacted By Code Modification*

These are the only tests impacted by the modification, meaning all other tests are irrelevant. The report (above) also illustrates this function is not fully covered by the test suite, leaving the following as our potential regressions:

1. A character not able to be concerted to Latin-1 remains untested (Unicode value greater than 255)
2. A null replacement character remains untested.

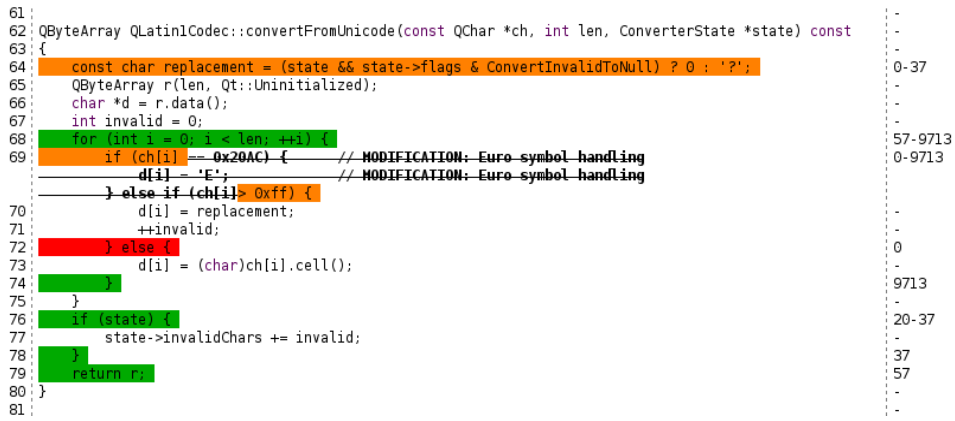


Figure 2: *Source Code View*

5.2 The Unit Test

The two missing unit tests are illustrated below: one for testing the € symbol conversion, and one for testing other Unicode values.
Modify `$QT5BASE/tests/auto/corelib/tools/qstring/tst_qstring.cpp` and define two new slots:

```
void euroSym();
void invalidUnicodeConversionToLatin1();
```

The slot code is:

```
void tst_QString::invalidUnicodeConversionToLatin1()
{
    QTextCodec *codec = QTextCodec::codecForName("Latin1");
    QVERIFY(codec != NULL);

    QString str;
    str += QChar(0x21AC);
    QByteArray latin1_converted = codec->fromUnicode(str);
    QCOMPARE(latin1_converted , QByteArray("?"));
}

void tst_QString::euroSym()
{
    QTextCodec *codec = QTextCodec::codecForName("Latin1");
    QVERIFY(codec != NULL);

    QString str;
    str += QChar(0x20AC);
    QByteArray latin1_converted = codec->fromUnicode(str);

    QCOMPARE(latin1_converted , QByteArray("E"));
}
```

Run the test suite again:

```
cd $QT5BASE
make -j$PROCESSORS
make -k check
find $QT5 -name "rcc.csexe" -exec rm -v {} \;
find $QT5 -name "moc.csexe" -exec rm -v {} \;
find $QT5 -name "uic.csexe" -exec rm -v {} \;
find . -name "tst_*.csexe" | while read LINE
do
    CSEXE="$LINE"
    CSMES=$(dirname "$CSEXE")/$(basename "$CSEXE" .csexe).csmes
    if [ -e "$CSMES" ]
    then
        echo -n "Generating␣Unit␣Test␣Database␣$CSMES␣...␣"
        cmcsexeimport -m "$CSMES" -e "$CSEXE" -t "unittest" -p merge
        echo "done"
    fi
    rm "$CSEXE"
done
for QTLIB in lib/lib*.csmes
do
    QTLIBTMP="$1".tmp
    UNITTESTS=$(find . -name "tst_*.csmes")
    echo -n "Importing␣Unit␣Tests␣Result␣in␣$QTLIB␣...␣"
    cmmerge -o "$QTLIBTMP"  -i "$QTLIB" $UNITTESTS
    mv -f "$QTLIBTMP" "$QTLIB"
    echo "done"
done
```

Following the test suite execution, the code coverage report contains the modification:

```
cd $QT5BASE
cmreport -m "$QT5BASE/lib/libQtCore.so.5.0.0.csmes" -s '.*' --html="$WORK/euro_sym.html" \
         --toc --title="Euro␣symbol␣support" \
         --source=all --method=all --global=all \
         -bargraph --source-sort=coverage --method-sort=coverage \
         -mr "$QT5REF/libQtCore.so.5.0.0.csmes" --execution=all
```

`$WORK/euro_sym.html` is generated, displaying the code coverage result for the modification.
The modification's coverage is 94%:

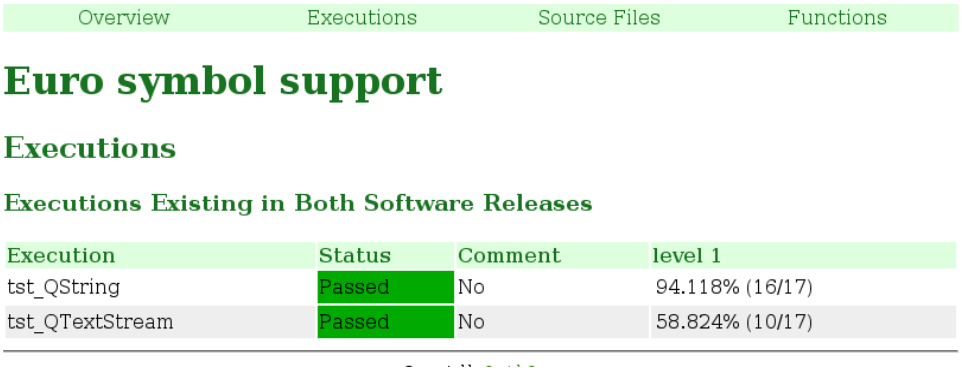
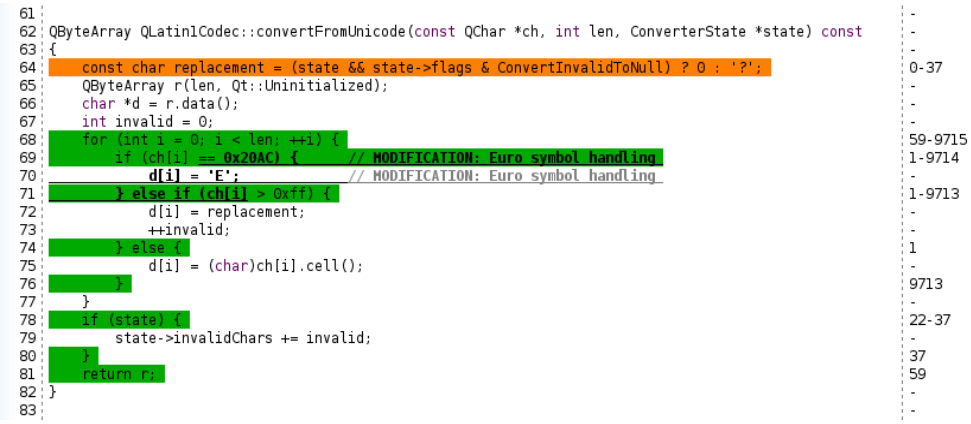


Figure 3: *Code Coverage Statistics Of The Code Modification*

The only line not covered corresponds to the missing test for selecting the replacement character:



```
61 |                                                                                  |  -
62 | QByteArray QLatin1Codec::convertFromUnicode(const QChar *ch, int len, ConverterState *state) const  |  -
63 | {                                                                                |  -
64 |     const char replacement = (state && state->flags & ConvertInvalidToNull) ? 0 : '?';  |  0-37
65 |     QByteArray r(len, Qt::Uninitialized);                                        |  -
66 |     char *d = r.data();                                                          |  -
67 |     int invalid = 0;                                                             |  -
68 |     for (int i = 0; i < len; ++i) {                                              |  59-9715
69 |         if (ch[i] == 0x20AC) {      // MODIFICATION: Euro symbol handling        |  1-9714
70 |             d[i] = 'E';             // MODIFICATION: Euro symbol handling        |  -
71 |         } else if (ch[i] > 0xff) {                                               |  1-9713
72 |             d[i] = replacement;                                                  |  -
73 |             ++invalid;                                                           |  -
74 |         } else {                                                                 |  1
75 |             d[i] = (char)ch[i].cell();                                           |  -
76 |         }                                                                        |  9713
77 |     }                                                                            |  -
78 |     if (state) {                                                                 |  22-37
79 |         state->invalidChars += invalid;                                          |  -
80 |     }                                                                            |  37
81 |     return r;                                                                    |  59
82 | }                                                                                |  -
83 |                                                                                  |  -
```

Figure 4: *Source Code View*

6 Conclusion

Using *Squish Coco* it is possible to reduce the code coverage analysis to only what the contributor and the approver needs:

- Is the patch correctly tested?
- What are the possible regressions?
- What is the impact, in terms of the number of tests, for the modification?

Of course, such analysis should be made by the integration server and performed on all Qt modules, integrated with the code review tool; however, this is topic for another day…

In order to help the Qt community to improve the quality, froglogic GmbH publish now code coverage statistics of QtBase computed on the daily snapshot. Once the official Qt5 will be available, we will also provide the code coverage analysis on the modifications between the actual snapshot and the last stable release.
Here the link: `http://download.froglogic.com/public/qt5-squishcoco-report`

7 Script Files

The following script was tested on Debian and performs the steps described in this article automatically.
To execute:

1. Copy `qt5-coco.sh` and `qt5-euro.patch` into a folder.
2. Execute `./qt5-coco.sh`
3. The generated files appear within `$HOME/coco`

Be patient: compiling and executing a Qt test suite as well as instrumenting it can be time consuming.

7.1 Build Script

```
#!/bin/bash
PATCH_FILE=$PWD/qt5-euro.patch
WORK=$HOME/coco
QT5=$WORK/qt5
QT5BASE=$QT5/qtbase
QT5QUICK=$QT5/qtquick3d
QT5DECLARATIVE=$QT5/qtdeclarative
QT5REF=$QT5/../qt5_lib_ref/
PROCESSORS=$(cat /proc/cpuinfo | grep processor | wc -l)

if [ ! -e "$PATCH_FILE" ]
then
    echo "Error: patch file $PATCH_FILE does not exists"
    exit
fi

function cleanup()
{
    cd $QT5 && git clean -xfd
    cd $QT5 && git submodule foreach --recursive 'git clean -dfx'
}

function checkout()
{
     if [ ! -e "$QT5" ]
     then
         cd $WORK
         git clone git://gitorious.org/qt/qt5.git
         cd $QT5 || exit
         ./init-repository -f
     fi
     cd $QT5 || exit
     git pull
     git submodule update
}

function configure_build_and_run_tests()
{
    cd $QT5 || exit

    ./configure -no-qpa-platform-guard -platform  linux-g++-64 -developer-build \
                -no-gtkstyle -no-pch -opensource -testcocoon \
                -confirm-license -prefix $QT5BASE || exit
    cd $QT5BASE
    make -j$PROCESSORS

    # executing test suite: -k is necessary to not stop at the first test
    make check -k
}

function process_csexe()
{
    CSEXE="$1"
    CSMES=$(dirname "$CSEXE")/$(basename "$CSEXE" .csexe).csmes
```

```
    CSEXE="$CSEXE"
    CSMES="$CSMES"
    if [ -e "$CSMES" ]
    then
        echo -n "Generating Unit Test Database $CSMES ... "
        cmcsexeimport -m "$CSMES" -e "$CSEXE" -t "unittest" -p merge
        echo "done"
    fi
    rm "$CSEXE"
}

function merge_unit_tests()
{
    QTLIB="$1"
    QTLIBTMP="$1".tmp
    UNITTESTS=$(find . -name "tst_*.csmes")
    QTLIB="$QTLIB"
    echo -n "Importing Unit Tests Result in $QTLIB ... "
    cmmerge -o "$QTLIBTMP"  -i "$QTLIB" $UNITTESTS
    mv -f "$QTLIBTMP" "$QTLIB"
    echo "done"
}

function collect_coverage_data_for_each_qt_library()
{
    cd $QT5BASE || exit

    # Process all execution reports of all unit tests
    find . -name "tst_*.csexe" | while read LINE
    do
        process_csexe "$LINE"
    done

    # Merge the result of the unit tests into the Qt libraries
    for QTLIB in lib/lib*.csmes
    do
        merge_unit_tests "$QTLIB"
    done
}

mkdir -p "$WORK"
rm -rf $QT5REF

checkout
cd $QT5BASE && git checkout -f

# compile
cleanup
configure_build_and_run_tests
collect_coverage_data_for_each_qt_library

# copy the instrumentation of the reference version of Qt
cd $QT5BASE
mkdir -p $QT5REF
mv $QT5BASE/lib/*.csmes $QT5REF/
```

```
# contribution
if [ ! -e "$PATCH_FILE" ]
then
    echo "$PATCH_FILE : file not existing"
    exit
fi
cd $QT5BASE || exit
cat "$PATCH_FILE" | patch -p1

# compiling, executing the suite after applying the changes
cleanup
configure_build_and_run_tests
collect_coverage_data_for_each_qt_library

# Code coverage report of the patch
cmreport -m "$QT5BASE/lib/libQtCore.so.5.0.0.csmes" -s '.*' --html="$WORK/euro_sym.html" \
         --toc --title="Euro symbol support" \
         --source=all --method=all --global=all \
         -bargraph --source-sort=coverage --method-sort=coverage \
         -mr "$QT5REF/libQtCore.so.5.0.0.csmes" --execution=all

# generate statistics for the whole Qt code
cd $QT5BASE
for QTLIB in $QT5REF/lib*.csmes
do
    HTML=$WORK/$(basename "$QTLIB" .csmes).html
    TTTLE=$(basename "$QTLIB" .csmes)
    cmreport -m "$QTLIB" -s '.*' --html="$HTML" --toc --title="$TITLE" \
             --source=all --method=all --global=all --dead-code \
             -bargraph --source-sort=coverage --method-sort=coverage
done
```

7.2 Patch

```
From 360e837828c244e3aaa92f794d7996c16a09703d Mon Sep 17 00:00:00 2001
From: =?UTF-8?q?S=C3=A9bastien=20Fricker?= <fricker@froglogic.com>
Date: Mon, 6 Aug 2012 18:42:56 +0200
Subject: [PATCH 2/2] Euro modif

---
 src/corelib/codecs/qlatincodec.cpp               |    4 ++-
 tests/auto/corelib/tools/qstring/tst_qstring.cpp |   25 ++++++++++++++++++++++
 2 files changed, 28 insertions(+), 1 deletions(-)

diff --git a/src/corelib/codecs/qlatincodec.cpp b/src/corelib/codecs/qlatincodec.cpp
index cf7c106..bb1ee5e 100644
--- a/src/corelib/codecs/qlatincodec.cpp
+++ b/src/corelib/codecs/qlatincodec.cpp
@@ -66,7 +66,9 @@ QByteArray QLatin1Codec::convertFromUnicode(const QChar *ch, int len, ConverterS
     char *d = r.data();
     int invalid = 0;
     for (int i = 0; i < len; ++i) {
```

```
-        if (ch[i] > 0xff) {
+        if (ch[i] == 0x20AC) {      // MODIFICATION: Euro symbol handling
+            d[i] = 'E';             // MODIFICATION: Euro symbol handling
+        } else if (ch[i] > 0xff) {
             d[i] = replacement;
             ++invalid;
         } else {
diff --git a/tests/auto/corelib/tools/qstring/tst_qstring.cpp b/tests/auto/corelib/tools/qstring/tst_qs
index 7476fa0..c17b41c 100644
--- a/tests/auto/corelib/tools/qstring/tst_qstring.cpp
+++ b/tests/auto/corelib/tools/qstring/tst_qstring.cpp
@@ -78,6 +78,8 @@ public:
 public slots:
     void cleanup();
 private slots:
+    void euroSym();
+    void invalidUnicodeConversionToLatin1();
     void fromStdString();
     void toStdString();
     void check_QTextIOStream();
@@ -245,7 +247,29 @@ private slots:

 template <class T> const T &verifyZeroTermination(const T &t) { return t; }

+void tst_QString::invalidUnicodeConversionToLatin1()
+{
+    QTextCodec *codec = QTextCodec::codecForName("Latin1");
+    QVERIFY(codec != NULL);
+
+    QString str;
+    str += QChar(0x21AC);
+    QByteArray latin1_converted = codec->fromUnicode(str);
+    QCOMPARE(latin1_converted , QByteArray("?"));
+}
+
+void tst_QString::euroSym()
+{
+    QTextCodec *codec = QTextCodec::codecForName("Latin1");
+    QVERIFY(codec != NULL);
+
+    QString str;
+    str += QChar(0x20AC);
+    QByteArray latin1_converted = codec->fromUnicode(str);
+
+    QCOMPARE(latin1_converted , QByteArray("E"));
+}
+
 QString verifyZeroTermination(const QString &str)
 {
     // This test does some evil stuff, it's all supposed to work.
--
1.7.2.5
```