Testing with Qt / QtCreator

Francesco Lamonica - 30/4/2021

Testing

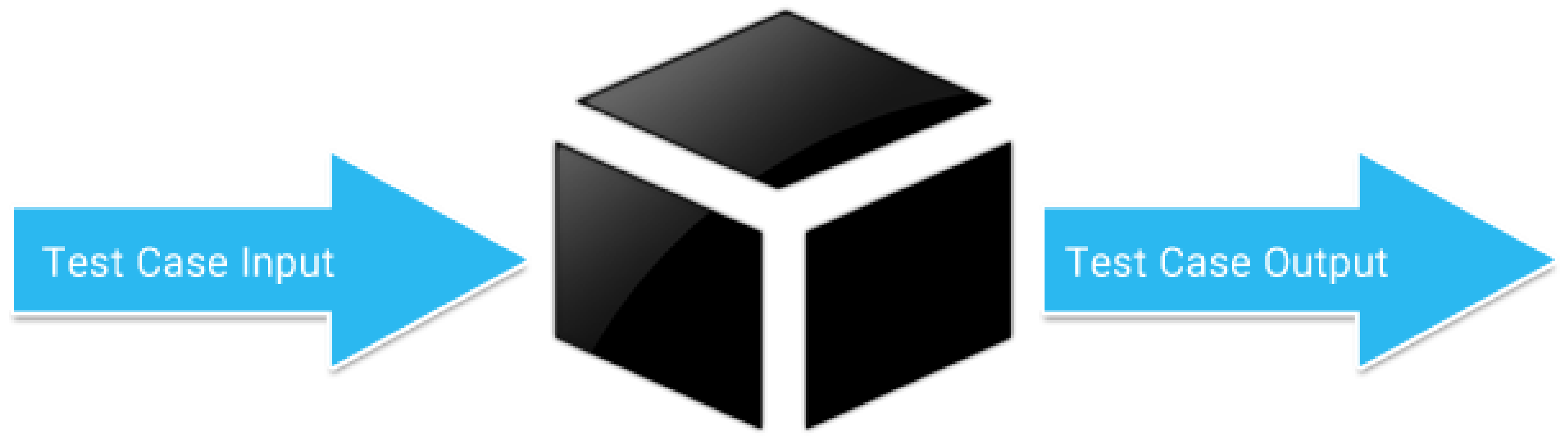
Unit, System, Integration, Validation… whatever

- There are many kind of testing: (wikipedia)
  - unit: In computer programming, **unit testing** is a software testing method by which individual units of source code—sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures—are tested to determine whether they are fit for use.
  - integration: is the phase in software testing in which individual software modules are combined and tested as a group. Integration testing is conducted to evaluate the compliance of a system or component with specified functional requirements
  - system / validation: **System testing** is testing conducted on a complete integrated system to evaluate the system's compliance with its specified requirements.

Testing

schema

- At the end of the day whatever test (unit, integration, system) you are doing is basically:



- Provide the [function|module|system] a known input and confront the output with the expected one

Testing environment

Enter QtCreator…

- To program in C++ we normally use QtCreator but this IDE can do much more than just edit / debug C++ files

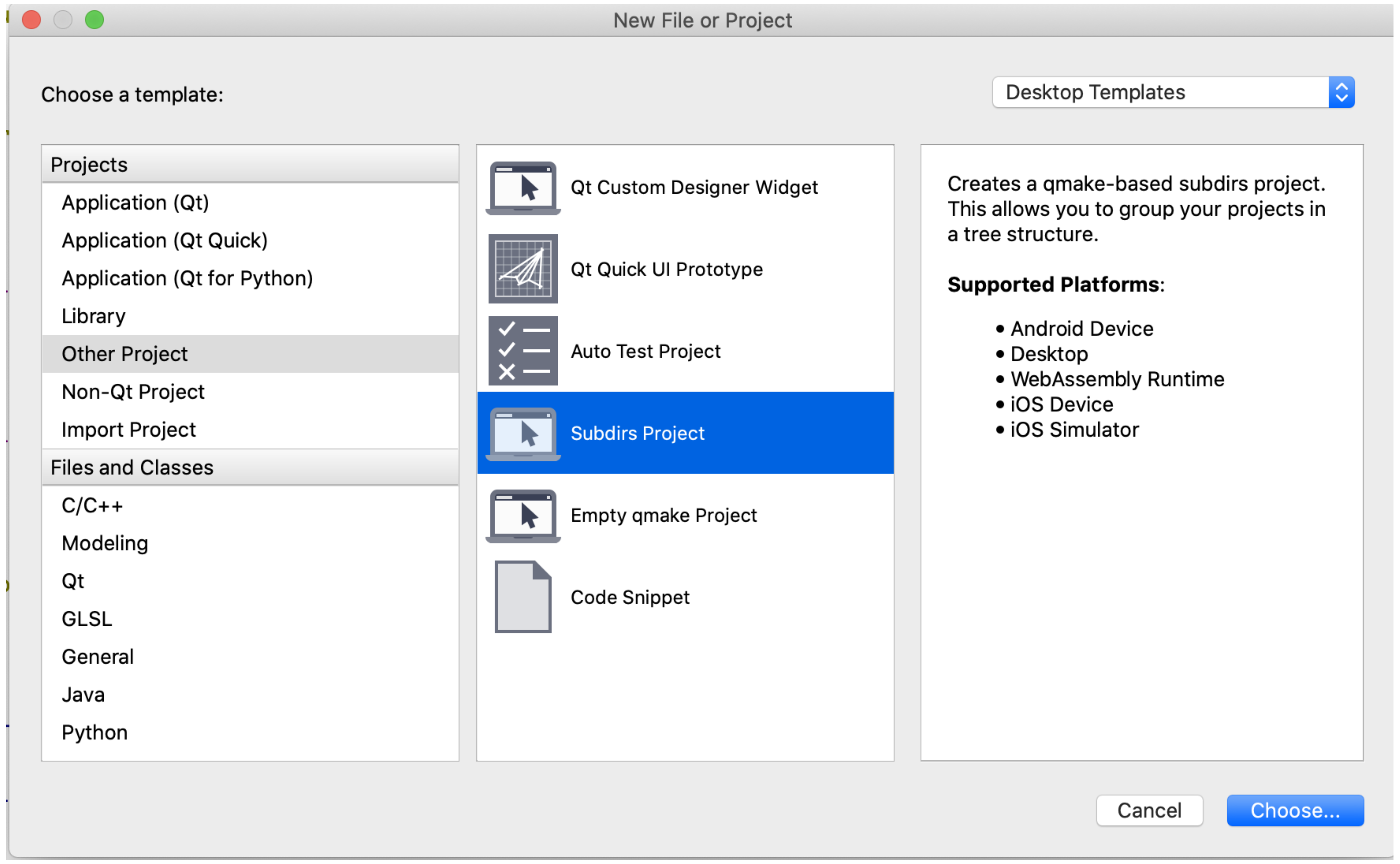
- Setup a testing environment for our project

- Run tests to assure that we did not introduce any regression

Testing with QtCreator

SUBDIRS project

- First of all in the same toplevel repository of your project create a new project of type SUBDIRS called “tests” or “ut”



New File or Project

Choose a template: Desktop Templates

Projects: Application (Qt), Application (Qt Quick), Application (Qt for Python), Library, Other Project, Non-Qt Project, Import Project

Files and Classes: C/C++, Modeling, Qt, GLSL, General, Java, Python

Qt Custom Designer Widget, Qt Quick UI Prototype, Auto Test Project, Subdirs Project, Empty qmake Project, Code Snippet

Creates a qmake-based subdirs project. This allows you to group your projects in a tree structure.

Supported Platforms:

- Android Device
- Desktop
- WebAssembly Runtime
- iOS Device
- iOS Simulator

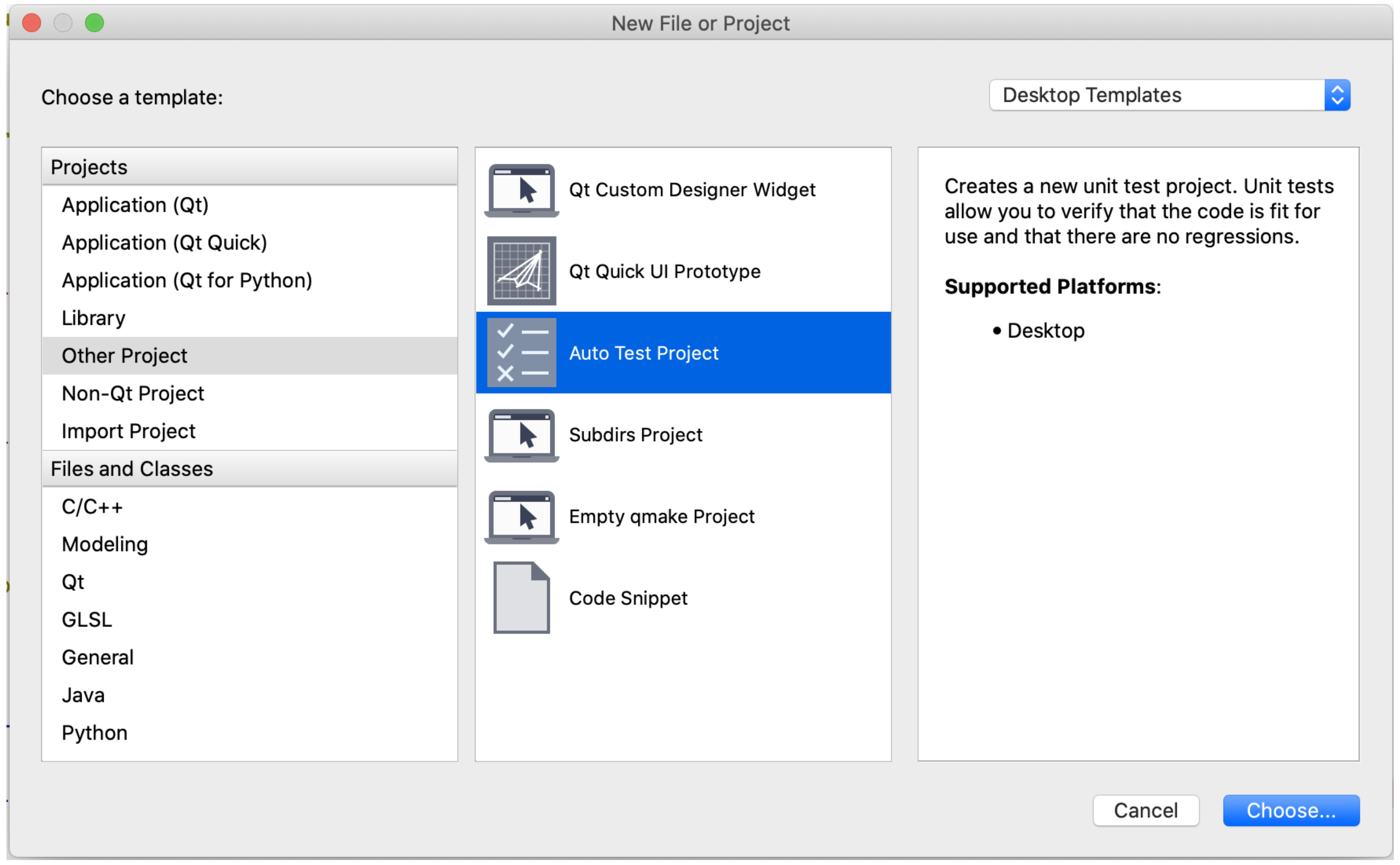
Cancel Choose...

```
TEMPLATE = subdirs

SUBDIRS += \
    TestTimeUtilsTicking \
    TestFileWriterRotationScenarios
```

Testing with QtCreator

The Auto test project

- Then, for each class / scenario / test case you want, create an AutoTest Project



Creates a new unit test project. Unit tests allow you to verify that the code is fit for use and that there are no regressions.

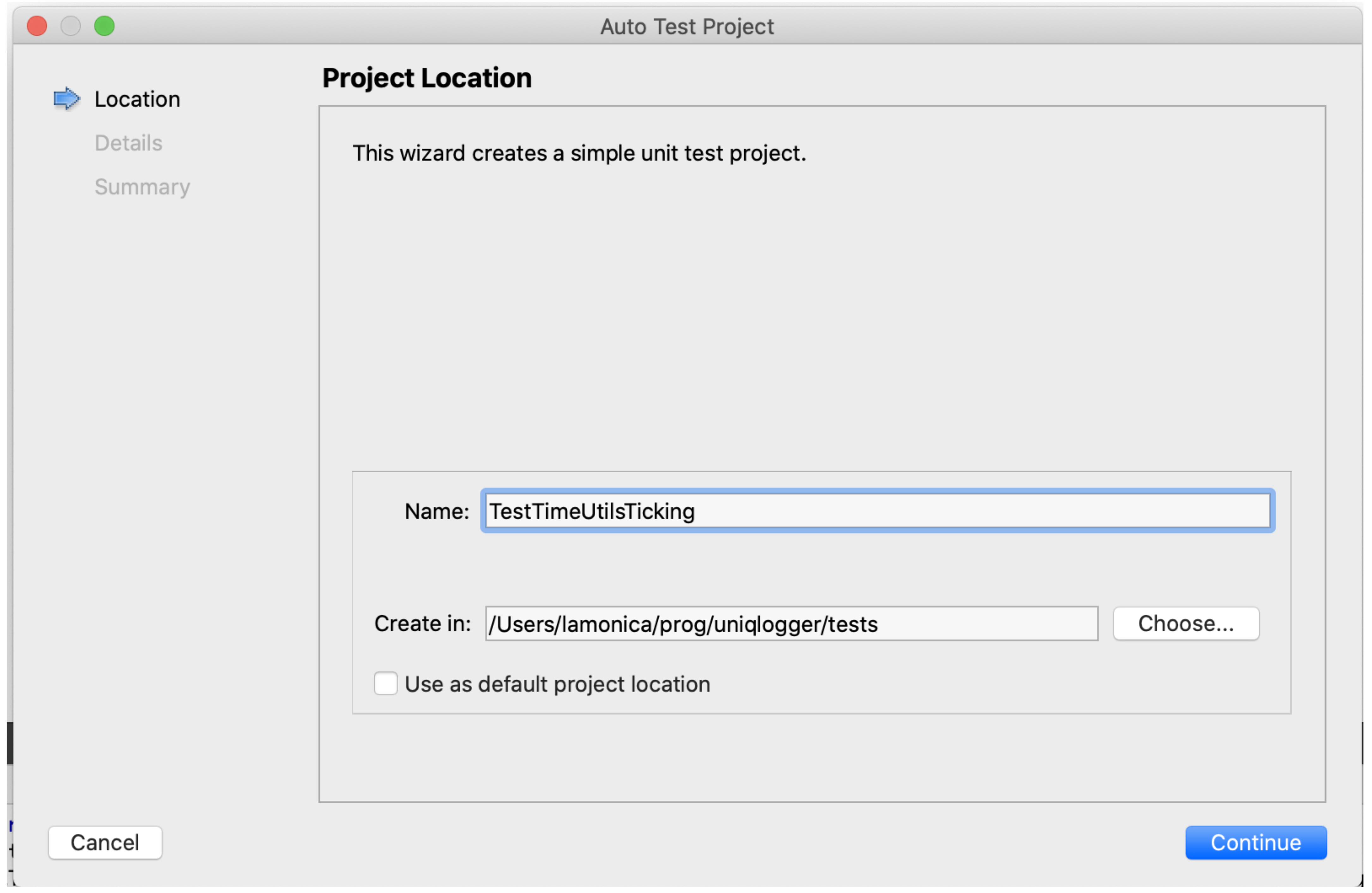
Supported Platforms:

- Desktop

Testing with QtCreator

The Auto test project - 2

- And choose the “tests” project folder you created earlier



Auto Test Project

Project Location

This wizard creates a simple unit test project.

Name: TestTimeUtilsTicking

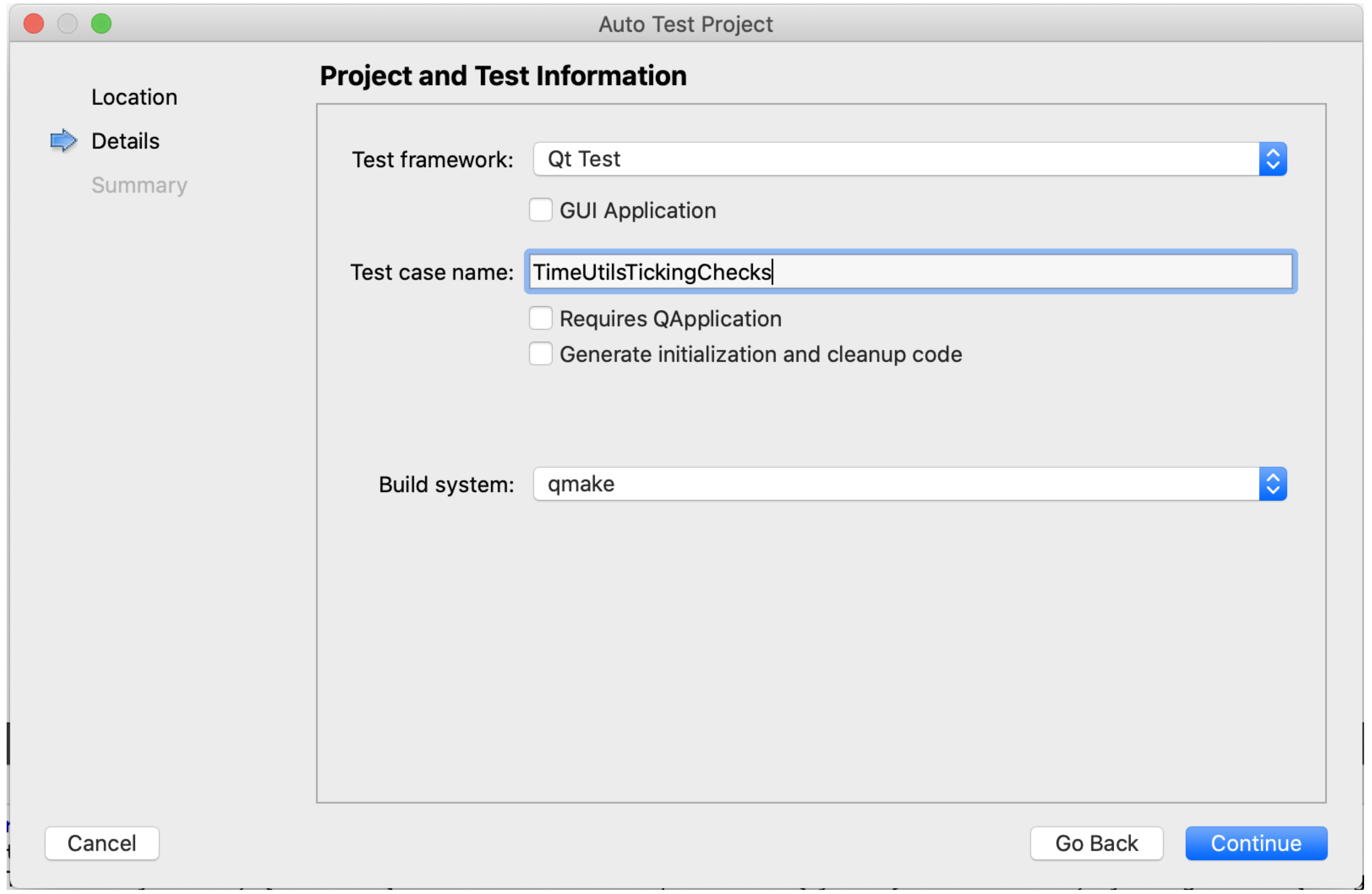
Create in: /Users/lamonica/prog/uniqlogger/tests

Use as default project location

Testing with QtCreator

The Auto test project - 3

- Then choose the name of the class that will store your test cases
- check whether the test cases will need either GUI or Application



Project and Test Information

Test framework: Qt Test

GUI Application

Test case name: TimeUtilsTickingChecks

Requires QApplication

Generate initialization and cleanup code

Build system: qmake

Testing with QtCreator

The Auto test project - 4

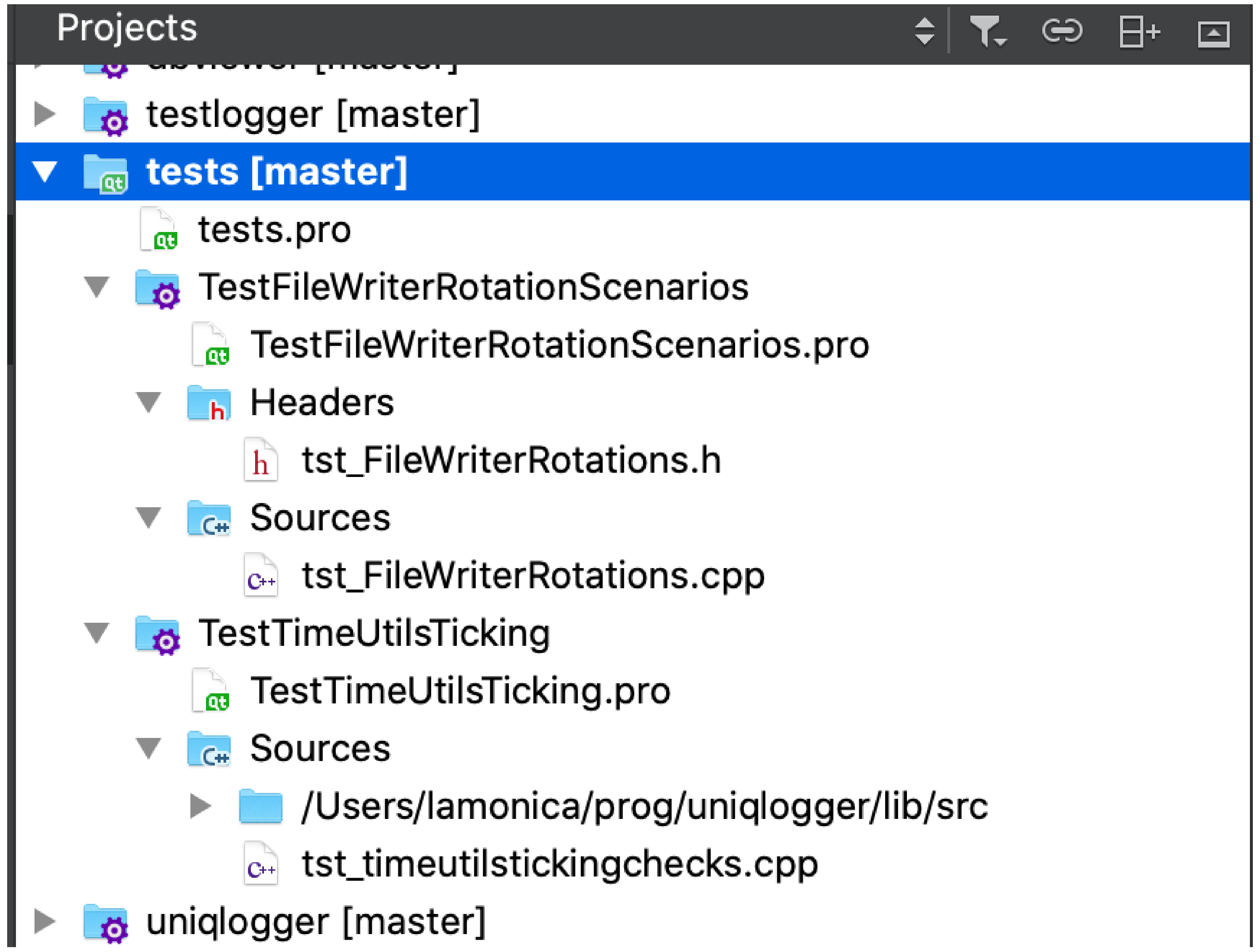
- The resulting .pro will be like this:

```
QT += testlib
QT -= gui

CONFIG += qt console warn_on depend_includepath testcase
CONFIG -= app_bundle

TEMPLATE = app

UNQLPATH = $$PWD/../../lib

INCLUDEPATH += $$UNQLPATH/src

SOURCES +=  tst_timeutilstickingchecks.cpp \
            $$UNQLPATH/src/TimeUtils.cpp
```

Testing with QtCreator

The project structure

- Eventually you will end with a structure similar to this



Projects

testlogger [master]
tests [master]
tests.pro
TestFileWriterRotationScenarios
TestFileWriterRotationScenarios.pro
Headers
tst_FileWriterRotations.h
Sources
tst_FileWriterRotations.cpp
TestTimeUtilsTicking
TestTimeUtilsTicking.pro
Sources
/Users/lamonica/prog/uniqlogger/lib/src
tst_timeutilstickingchecks.cpp
uniqlogger [master]

Testing with QtCreator

Conventions

- Auto tests files should start with “tst_” prefix
- All test cases should be defined as “private slots” methods in test classes in order to be executed automatically
- Each auto test project should test one class (but multiple test cases)
- This class should be called within one of the 3 macros:
  - QTEST_APPLESS_MAIN(YourTestClassName) - to test GUI-less classes that do not need QCoreApplication
  - QTEST_GUILESS_MAIN(YourTestClassName) - to test console-based apps/classes
  - QTEST_MAIN(YourTestClassName) - to test graphical classes

Testing with QtCreator

Useful macros

- QSKIP(ReasonString): put in a test and it will be skipped printing the ReasonString in the report
- QVERIFY(boolean condition): this is one of the 2 main macro to be used to test that your test is doing what is supposed to do: that the boolean condition is true
- QCOMPARE(val1, val2): compares two values and fails if they are different, the main advantage over QVERIFY(val1==val2) is that the two values are printed
- QBENCHMARK{ CODE BLOCK }: will write in the report the time spent in that block

Testing with QtCreator

What now?

- Ok we have created our subdirs project and all the auto-tests sub-projects, and now?

- Go to the tests folder and type:
  - make check

- All the auto-tests will be run and verified automatically and the summary report will be printed in console (or within QtCreator)

Testing with QtCreator

An example from the docs

```
#include <QtTest/QtTest>
class TestQString: public QObject
{
    Q_OBJECT
private slots:
    void toUpper();
};

void TestQString::toUpper()
{
    QString str = "Hello";
    QCOMPARE(str.toUpper(), QString("HELLO"));
}
QTEST_MAIN(TestQString)
#include "testqstring.moc"
```

```
        Start          TestQString
Config: Using QtTest library %VERSION%, Qt
%VERSION%
PASS   : TestQString::initTestCase()
PASS   : TestQString::toUpper()
PASS   : TestQString::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
        Finished          TestQString
```

How do you write a test?

Writing Tests

best practice - 1

- Writing Unit and Integration / System tests can be different:
    - Unit tend to be self-contained, you only need the class that holds the function you want to test and you’re good to go
    - System tests means that you want to test a feature end-to-end this (in terms of the auto-test project) result in a .pro file that will certainly include many classes, possibly linking dependent libraries to create an executable that provides the feature we want to test

Writing Tests

best practice - 2

- To better deal with System tests when creating a project always split your .pro file in two:
    - a .pri file that includes all the project
        - sources,
        - headers,
        - linking commands to dependencies
    - a .pro file that includes:
        - the above .pri file
        - main.cpp
        - versioning and other application specific configurations

Writing Tests

best practice - 3

- Tests should be run often (at every build eventually) so keep them small and lean: Much much better to create multiple tests that you can eventually skip if needed
- Tests should be idempotent: system tests may alter the environment (i.e. write a file) always cleanup after you are done so that you are always in control of all of your inputs and outputs
- Don’t Repeat Yourself: if you need some “plumbing” code to enable your tests, by all means put it in a TestClass::method and call it when needed (don’t put it in your test code)

Writing Tests

best practice - 4

- In order to allow easier setup/tear-down of tests Qt offers the following methods (to be declared as “private slots”)
- initTestCase() will be called before the first test function is executed.
- init() will be called before each test function is executed.
- cleanup() will be called after every test function.
- cleanupTestCase() will be called after the last test function is executed.

Writing Tests

best practice - 5

- In order to allow easier setup/tear-down of tests Qt offers the following methods (to be declared as “private slots”)
  - initTestCase() will be called before the first test function is executed.
  - init() will be called before each test function is executed.
  - cleanup() will be called after every test function.
  - cleanupTestCase() will be called after the last test function is executed.

Writing Tests

best practice - 6

- Don’t stick to the “happy path” fuzz your inputs and test that your [function|system|whatever] remain coherent
- Remember the sample toUpper() test ? What if we want to test multiple inputs?
- Qt Offers the “*_data()” methods to ease the repetition of a test against multiple inputs

```
#include <QtTest/QtTest>

class TestQString: public QObject

{

   Q_OBJECT

private slots:

   void toUpper();

};


void TestQString::toUpper() {

   QCOMPARE(“HEllo”, QString("HELLO"));

   QCOMPARE(“hellO”, QString("HELLO"));

   QCOMPARE(“HeLlO”, QString("HELLO"));

}

QTEST_MAIN(TestQString)

#include "testqstring.moc"
```

Writing Tests

best practice - 7

- First refactor toUpper() using the QFETCH macro to get some generic “StringToTest” and “ExpectedResult” both of type String
- Then implement the toUpper_data() private slot that will help us populate those generic variable with multiple data
- toUpper() thanks to the QFECTH macro will be tested against all possible inputs

```
#include <QtTest/QtTest>
class TestQString: public QObject
{
   Q_OBJECT
private slots:
   void toUpper();
};
void TestQString::toUpper_data() {
   QTest::addColumn<QString>("StringToTest");
   QTest::addColumn<QString>("ExpectedResult");

   QTest::newRow("all lower") << "hello" << "HELLO";
   QTest::newRow("mixed")     << "Hello" << "HELLO";
   QTest::newRow("all upper") << "HELLO" << "HELLO";
}
void TestQString::toUpper() {
   QFETCH(QString, StringToTest);
   QFETCH(QString, ExpectedResult);
   QCOMPARE(StringToTest, ExpectedResult);
}
QTEST_MAIN(TestQString)
#include "testqstring.moc"
```

Writing Tests

best practice - 8

- Test should be isolated and not depending on other tests or internal states
- Should i write “C” code?
- Leverage C++ constructs
  - Dependency Injection / Factories / inheritance / special accessors
  - How to test protected functions?
  - How to test private functions?

UniqLogger use-case

UniqLogger use case

The size-based file rotation

- UniqLogger already had the size-based file rotation that would store log messages over a configurable number “n” files switching to a new one when the current reached the maximum size that was configured
- What happens after we reached logging to “n” files is defined by a policy
  - strictrotation (similar to logrotate)
  - incremental number

UniqLogger use case

The size-based file rotation - 2

- strictrotation (similar to logrotate)
  - log.txt will always hold the most recent logs
  - log-1.txt will hold the slightly older ones
  - log-n.txt will be storing the oldest logs
  - each time log.txt reaches the max size the “log-n.txt” is scrapped and all the others renamed accordingly: i.e. log-1.txt -> log-2.txt
  - a new log.txt is started

UniqLogger use case

The size-based file rotation - 3

- incremental number
    - log.txt will always hold the oldest logs
    - log-X.txt will be storing the newest logs
    - each time log-X.txt reaches the max size:
        - all the oldest log files are scrapped (up to log-(X-n).txt)
        - a new log-(X+1).txt is started
    - this is more performant since there is no moving around all the old files

UniqLogger use case

The size-based file rotation - 4

- Did i hear someone mentioning ZIP?

- Either policy can be also configured to (g)zip the other than most-recent log file to save space

- We have a lots of possible use-cases: log-n.gz, log-n.zip, etc.

UniqLogger use case

Enters the time-based file rotation

- When you have a production environment, defects are reported with the time when they occurred, if the log files are rotated just on a size basis two things can happen:
  - the log messages were too fast and fill up the number of files that were configured -> you lost your logs!
  - you have configured a big-enough size for the log files but it could be cumbersome to analyse a file big hundreds of MB
  - we need a time-based rotation

UniqLogger use case

time-based file rotation goals

- Obviously the time-based rotation should sit “on top” of other size-based rotation because there can be environments (kalliope, Atena bots, etc.) where the size constraint might be mandatory
- So i decided to allow: Day, Hour and minute (mostly for development) rotation policies that would switch log file whenever a new [day|hour|minute] “ticks”

UniqLogger use case

time-based file rotation policies

- So i decided to allow: Day, Hour and minute (mostly for development) rotation policies that would switch log file whenever a new [day|hour|minute] “ticks”

- Upon suggestion there was another possible policy: elapsed time
    - this would trigger the switch to new file whenever the time elapsed not when the threshold (day, hour) “ticked”.

UniqLogger use case

time-based file rotation examples

- HourlyRotation
  - if we started logging at 2021-04-30T17:58:34 we would have
  - initial logfile: log-2021-04-30T17:58:34.txt and we would switch to the new file when the hour ticks: log-2021-04-30T18:00:00.txt
  - in this case the initial log file will be storing just 26 minutes worth of logs.
  - next log file will be log-2021-04-30T19:00:00.txt

UniqLogger use case

time-based file rotation examples - 2

- HourlyRotation with strict size-based rotation (3 files of max 10MB each)
  - if we started logging at 2021-04-30T17:58:34 we would have
  - initial logfile: log-2021-04-30T17:58:34.txt
  - suppose we write less than 10MB in 26min
    - would switch to the new file when the hour ticks: log-2021-04-30T18:00:00.txt
  - now suppose we write more than 10MB in 40min
    - current log file will be renamed (and maybe zipped) to log-2021-04-30T18:00:00-1.zip
    - new log file will be still log-2021-04-30T18:00:00.txt and would be holding logs starting from 2021-04-30T18:40:00

UniqLogger use case

time-based file rotation examples - 3

- [Daily|Hourly|PerMinute]Rotation with [strict|incremental] size-based rotation would be following the same patterns

- If you are starting to think that it seems a bit difficult to get it right you are smarter than me

- I started over 3 times and the last one i decided that i needed tests!

Let’s go to the code