swWxGuitTesting

Unmodified Thesis Chapter by Reinhold Füreder

Contents

List of Figures

List of Listings

Chapter 1

Automated Testing

In this chapter automated testing of code units, as well as GUIs and whole applications – in the following applications in the context of testing always stands for GUI based applications in opposition to command line applications – is first introduced and its state of the art reviewed. Then requirements and implemented solutions of these approaches applied on the developed tools and application are described in detail, making up an important part of this thesis. Also, experiences gained in the course of introducing testing in the shape of a test-first software development oriented methodology to the project are stated.

1.1 Introduction

Broadly speaking, software testing serves the purpose of revealing the *correctness* of code or applications. It can be split into validation versus verification. Binder, one of the most acknowledged experts in software testing, states: "Validation asks the question 'Has the right thing been built?' 'Is the specification correct?' or 'Are all user requirements met?' Verification asks the question 'Has the thing been built right?' 'Does the implementation fully and correctly realize the specification?' or 'Do all components exhibit good workmanship, sufficient performance, and conform to applicable standards?'" [Bin01] Thus, *verification* checks the accordance of software products with their specification and is employing test cases; *validation* evaluates the benefit or value of the product for the user [Bal98].

In the following Pol's definition for testing, which clearly classifies verification and validation as part of the theory of testing, and testing as such as being part of the practice of testing, is used [PKS02]: *Testing* is executing the code with a limited, but representative amount of test data, and comparing the expected results with the actually computed results. So, the correctness of code cannot be proven by testing. By contrast, verification uses any potential input data (and its combinations) and therefore allows checking the consistency of code and code specification. As

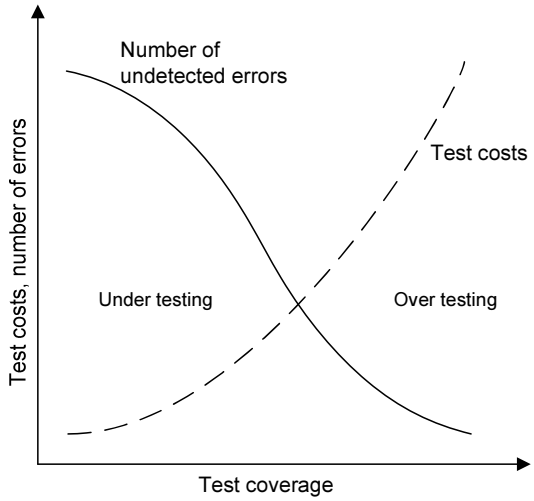


Figure 1.1: Test costs versus number of undetected errors.

a consequence of including cost effectiveness into the requirements of testing Myers [Mye79] rejects verification due to being infeasible.

Testing cannot show or guarantee that code is *error-free*, but by aiming at finding as many errors as possible (not necessarily all) it increases the confidence in the quality of the written software. Hence, testing should be carried out as long as the budget is allowing it, and it makes economically sense in that the costs of finding and removing errors during testing are less than the costs of detecting errors in the course of software product usage by clients or customers (see Figure 1.1). Therefore pragmatic programming advises to concentrate the testing effort on critical parts of the code, i.e. error-prone code, complex code, or code with severe consequences in case of errors [HT99].

While the aforementioned holds true for testing independent of its degree of automation, from now on testing refers to automated testing in opposition to manual testing.

1.1.1 Testing Notions

The following paragraphs deal with basic concepts and the most important terms required for the intended pragmatic testing approach used in the agile development methodology. For a more thorough and extensive essay it is referred to Binder's "software testing bible" [Bin01].

A test object is the object being tested, or in other words, the software or programme unit under test. By analogy, the terms implementation under test (IUT), unit/class under test (UUT), component under test (CUT), system/application under test (SUT), and so on, are widely used.

The basic elements of testing are *test cases* which consist of test data, i.e. input and expected output (to be produced by the IUT from the input), and executable test routines (procedures). Executing or running a test case is equated with running a test. *Test suites* are combining one or more test cases in order to structure them logically. Running a test suite means running all its

tests.

Test environments (or test beds) are containers for holding and running test cases and provide the infrastructure for test system. Another important part of test beds are *test drivers* which are little utility programs actually initiating the execution of tests.

Testing code pieces in isolation from the system requires simulating (parts of) the system by *test stubs*. Thus, during test execution the test stub imitates the behaviour of code currently not under test. Dummy and mock are often synonymously used for stub, although strictly speaking, they are not the same.

1.1.2 Unit Testing

With eXtreme Programming (XP) a lightweight software development methodology has become very famous and one of its rules postulates unit testing, i.e. (functional) testing of smallest possible units of object-oriented (OO) code. Moreover, unit testing should be carried out in a test-first manner, or based on Test-Driven Development (TDD). Consequently, it is implemented by the software developer who is also developing the unit. Creating and executing these tests as early as possible in the development process makes the detection and removal of errors the cheapest. Another advantage is that localising emerged errors in the source code is enormously facilitated due to the small size of units under test.

Of course testing only small units, typically classes in Object-Oriented Programming (OOP), in isolation cannot demonstrate that their combination and collaboration leads to correct behaviour [FHJ+04]. Also, in the sense of OOP unit is a very ambiguous term: it can stand for a method, a class, several classes, a software subsystem, or even the whole software system or application. Therefore, it is differentiated between low-level and high-level units. From now on in this thesis unit testing is denoting not just the testing of a "traditional" OO class, but any unit at any level if not stated otherwise.

In analogy to low-level and high-level units, *test scope* is split into low-level and high-level tests respectively. Low-level tests test low-level units and comprise module (component, unit) and integration (interaction) tests. High-level tests test high-level units and thus, include system (application) and acceptance tests.

The latter involves the user or client to a great extent in that she is either defining (for automated acceptance tests), or actually manually performing the acceptance test. This serves the purpose of finding out if systems (or their milestones) really allow users the fulfilment of the intended goals, i.e. it answers the question "Does the user accept the system, that is, the system's functionality?".

See Figure 1.2 for a graphical classification of test categories with respect to (a) test and unit levels, as well as (b) to the isolation level.

Lastly, testing procedures are also categorised by the level of insight into the IUT. *Black-box testing*

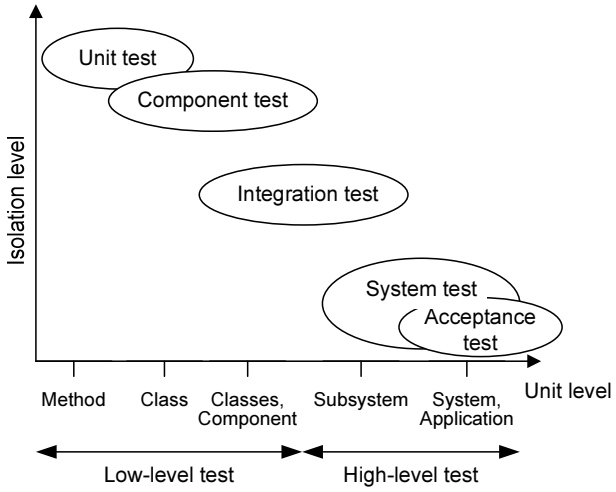


Figure 1.2: Classification of object-oriented unit tests.

looks at the specification to create tests, i.e. test data and test routines, while *white-box testing* carefully consults the actual code-internal data flow.

1.1.3 Integration Testing

The process of creating (sub-)systems by putting together (i.e. building relations and interactions) a number of small and in isolation tested units is called *integration*. Basically, two integration procedures are possible: incremental or big-bang integration. The latter (tries to) combine all units at once and is/was rather standard in "traditional" software development. By contrast, incremental integration is in particular pushed by agile development methodologies and is composing by adding one unit at a time. This allows (a) adding comparatively straightforward *integration tests*, and (b) running these tests after each little integration step. Accordingly, newly detected errors can only stem from the last little integration step, making the error localisation very easy in general.

Integration testing needs to test interactions between several classes. Again out of the aforementioned reasons the goal is making this in as little increments as possible. This typically requires the simulation of parts of IUT, as well as the simulation of classes with which the IUT interacts: stubs come into play.

A further step forward from incremental integration is Continuous Integration (CI), which, again, is forced by agile methodologies and in particular by XP. CI asks for regular (at least daily) committing of code, and comprises, depending on parameters like criticality, size of the integration step, frequency of code changes by other members, etc. either before and/or after committing, the following steps: (1) (clean) check out of code, (2) build code, and (3) run all tests. According to Fowler [FF] the more developers working on the same project, and the more geographically distributed the development team is, the more important is using CI.

Applied on ScanCAD several options for a potential implementation of CI are possible. Due to using CMake for building the application and all required Simpleware libraries *CTest* – as part of CMake – in combination with *Dart*, "an open-source, distributed, software quality system" [Kit] is a potentially good starting point because they can be easily configured for any existing CMake project file. The two tools provide numerous advanced features, for instance checking code coverage and memory errors, or a server dashboard (e.g. for ITK: `http://www.itk.org/Testing/Dashboard/MostRecentResults-Nightly/Dashboard.html`) where all test results are automatically published. Only in the case of introducing CTest and Dart late in the development process one serious disadvantage arises: they make some assumptions about how test cases and test drivers are designed and implemented.

To circumvent this problem another option could be a simple proprietary approach for the currently Microsoft Windows centric development based on simple batch files (or shell scripts if using cygwin for instance) and employing CMake:

1. Clean CVS check out of SimplewareFramework, SimplewareUtils and ScanCAD modules.
2. Clean build of all modules including tests (using XML test driver outputter for automatic post-processing) with CMake using the Microsoft NMake build generator.
3. Run all tests first in debug, and only if succeeding, in release mode; failing assertions can only be detected in debug mode, but would cause crashes due to access violations in release mode.

Finally, in [CM] Caputo and Miller present their implementation of CI in a Microsoft Windows/-COM environment for a project developing primarily in Microsoft Visual `C++` (MSVC). Although it is platform and even Integrated Development Environment (IDE) specific, it contains some interesting arguments about the importance of dependency management, besides employing Ant and Cruise Control, two well-known Java CI tools.

1.1.4 Regression Testing

As already pointed out running tests as often as possible, and after each little integration step is the most effective in getting and keeping confidence in the code, as well as localising and removing detected errors. Of course, this is even more important in incremental or agile development processes where a lot of rather little code refactorings (i.e. modifications, improvements) are likely to be carried out throughout the whole product development life cycle [Fow00].

This is nicely summarised in Cunningham's Portland Pattern Repository's Wiki [C$^+$]: "Regression testing is testing [...] that the functionality that was working yesterday is still working today."

1.1.5 Test-Driven Development

In 2002 Kent Beck has published a lightweight software development methodology built "around" testing, namely Test-Driven Development. Like XP it focuses on little increments and follows a test-first procedure, i.e. writing tests before writing code (= IUT).

As stated in [Bec02] its goal is "clean code that works" which is reached by designing organically, and adhering to two simple rules:

- Write new code only if an automated test has failed.
- Eliminate duplication.

These two rules demand a certain coding procedure:

1. Writing a little test which is doomed to fail, maybe it cannot even be compiled.
2. Making the test work quickly even accepting obviously poor and short-living solutions.
3. Eliminating the duplication (i.e. through applying refactoring) probably introduced by making the test work quickly.

The consequences are that the user is first thinking about how to test the code and by writing the test first, the interface of the code (which is written subsequently) is already predefined and clean. Moreover, running/working tests are equated to finished code. Finally, these short development cycles or iterations make the integration easy, and in particular the inherent refactoring as final part of adding features guarantees clean and unembellished code which does not "smell".

1.1.6 Automated GUI Testing

As aforementioned, supplementing low-level tests (i.e. unit testing and integration testing) with high-level tests is an important strategy for the purpose of fulfiling correctness requirements of the whole software system. Due to the fact that the developed tools and the application ScanCAD are heavily Graphical User Interface (GUI) based, automated GUI testing is mandatory. Manual GUI testing is too labour-intensive and monotonous in consideration of the huge number of possible controls (e.g. buttons, trees) and the various ways of interaction.

As listed on [Fau04] a huge number of GUI testing tools are around. Ranging from open-source to fairly expensive tools, mostly platform dependent or even only for one platform available (this contradicts the requirements from Section **??**) this list does rather increase the confusion about what tool should be chosen. Merely or in particular for Java based GUI applications several excellent open-source tools and toolkits are available: cf. jfcUnit [CAW04], Abbot [Wal04], Jemmy [Net], or Marathon [The04]. By contrast the market of existing free or open-source tools for applications using `C++` is pretty limited, or not existent.

A number of free GUI event recorders are available for individual platforms, thus they are principally platform dependent. Of course, their inherent nature of listening and interpreting platform specific low-level events makes them (programming) language independent. Unfortunately, they are unspecific to automated GUI testing in general, and are therefore lacking sophistication and user-friendliness in this respect. One of the closest example to automated GUI testing is the so-called MSDN Tester, developed by Robbins in the Bugslayer column of MSDN [Rob99, Rob00, Rob02]: due to its unreliability, requiring to restart the Operating System (OS) for instance, the limited usability and lack of features were not the real problem ...

Commercially available tools aimed at automating GUI testing are in most cases Capture & Replay (C&R) based, making the GUI testing easy and fast according to the manufacturers. The principal operation of a so-called Capture & Replay tool (CR-tool) is:

1. *Capture mode:* In analogy to GUI event or macro recorders any user interaction with the GUI (i.e. user input) is recorded and transformed into test code, often called test script.
2. *Checkpoints:* For the evaluation of the (functional or non-functional) SUT's correctness assertions or checkpoints are inserted into the test script.
3. *Replay mode:* Captured test scripts can be (re-)run for the purpose of GUI related regression testing; failing checkpoints in the course of replaying lead to test failures.

In order to guarantee independence of screen resolution, size and position of panels, etc. storing, for example, the coordinates of mouse clicks during capturing is not enough:

- The affected control must be uniquely identifiable on different machines with different visual software system properties.
- When some of a GUI control's properties like position are changing, the tool must be able to re-detect and re-recognise it; otherwise it leads to a maintenance nightmare.

Typically test scripts are written in high-level programming languages similar to BASIC or JavaScript, but often proprietary and less powerful. Nevertheless, these languages allow adding complex test routines and assertions in between the recorded test steps.

As already indicated, replaying should support different machines with different OS versions in order to maximise the test "coverage". This requirement is even harder in case of platform independent SUTs.

Recognising GUI objects is also part of the very first phase of replaying. In this connection GUI objects from different interfaces using non-natively rendered widgets, i.e. not provided by the underlying platform, can be insuperable hurdles. In addition, the actual GUI object recognition based on its ID (plus sometimes additional object properties) is often problematic since ID and/or properties are likely to change during development making the maintenance laborious.

Another inconvenience is linked with the way how the GUI is designed. As outlined in Section **??** and Section **??** XML-based resource files describing the GUI are created, with symbolic names uniquely identifying the individual controls. This allows accessing the controls programmatically at runtime. The problem now is that the CR-tool does not know about these XML-based resource (XRC) files, henceforth it cannot use these symbolic names directly. Again, in the best case one can assign them conveniently to the numeric IDs of detected GUI controls. Nevertheless some duplicate work has to be carried out.

Likewise, all kind of difficulties can occur during the replay phase because "test cases depend on the SUTs history and current state [..., e.g. GUI objects] are enabled/disabled depending on [...other objects, or] an email message popping up captures the focus and stops your test-run" [LD98]. Likewise unexpected exceptions and errors can lead either to pop-up message boxes stopping the test and demanding manual user interaction, or, if not caught, to crashing SUTs in the worst case.

IBM Rational Suite TestStudio [IBM], Segue Silktest [Seg05] or Winrunner by Mercury Interactive [Mer05] are well-known commercial tools providing additional features like reporting or test data management with databases to support the whole testing life cycle. On the other hand these tools come at prices from approximately €5,000 to €10,000.

Only recently (29 March 2005) a wxWidgets based CR-tool called Zephyr Automated Test Runner was released claiming "[...to be] an automated functional and regression testing tool [...which] will test any application, written in any language, that has a GUI interface." [Deg05]. Although the price of $699 sounds reasonably, not the limited features, but the tool's core method is the crux of the matter: all checks and verifications are based on primitive image comparisons.

Besides the potentially high price the handling of this tools by developers is equally unappealing: captured scripts need to be maintained and checkpoints added, and only in the best case the scripting language is not a proprietary development but standardised language like JavaScript, with which nevertheless some developers are not familiar.

As a consequence of the aforementioned hurdles of CR-tools Feathers and others postulate a pragmatic approach of testing the layer below the GUI [Fea02]: testing only the "interactions" of GUI controls based on mock view objects which necessitates a clean layering of the software system by means of the Model-View-Controller (MVC) pattern (cf. [GHJV94]) for instance. This rather straightforward method has some drawbacks:

- The GUI itself, or its behaviour is not really tested; only its assumed behaviour implemented in the shape of mock view objects, besides the interaction and collaboration of the remaining related objects under test.
- For each GUI control a corresponding, admittedly straightforward, mock view object must be implemented; this is of course a one time investment only, and typically the (limited)

number of standard GUI controls is by far exceeding the number of specialised ones, paying off this effort extremely soon.

To overcome these drawbacks which are also discussed on [C+04a], Plumlee et al. are suggesting a pragmatic self-developed, typically platform and GUI toolkit specific approach on the TestFirstUserInterfaces (TFUI) mailing list [P+] and on [C+04c]. Its main pattern is called "Then Don't Call Main Loop" [C+04d], suggesting to interact with the GUI as normally (e.g. creating windows, putting data in them, reading data out), but without calling or running the main loop to receive events and dispatch controls' callbacks.

Finally, an interesting approach to reduce the time-consuming (manual) creation of automated GUI tests by automating the generation (of the bulk) of test cases is presented in [MPS00]. This allows not just testing one of the various GUI control specific interactions, but the most likely ones. The automated inclusion of common checkpoints or test assertions – defined in databases for example – by means of declarative GUI control specific options could make the test case generation for a GUI really a matter of minutes rather than hours.

This approach partly mitigates some of Kaner's concerns, who notes that GUI regression testing is not automated, but computer assisted testing [Kan02]: "It doesn't automate very much of the testing process at all." This leads to high prioritised GUI testing requirements: pragmatic, maintainable, and relatively effortless.

1.2 Aims & Requirements

After introducing automated testing and discussing especially the state of the art with respect to automated GUI testing, project specific aims and requirements of automated testing are described in this section.

As pointed out in Section **??**, owing to the industrial sponsorship of this work special emphasis is put on robustness, correctness and testability. The developed tools and application are partly algorithm intensive and partly GUI centric, thus algorithm (unit) and GUI testing are equally important. Due to limited resources (personal and financial) the purchase of any advanced commercial CR-tool is not the best option: the time and effort to use it effectively is quite big, as it always involves new and mostly even proprietary techniques. Instead, and considering the underlined evolvability, a rather pragmatic approach fitting to the employed agile development methodology as stated in Section **??** is the ultimate goal.

The sub-goals of automated testing comprise:

1. *Unit/non-GUI-integration testing:* Testing functionality of (a) single units or classes, and (b) non-GUI related collaborations of class compounds based on the clean separation and layering by means of the MVC pattern.

2. *GUI testing:* Testing (a) functionality, and (b) correct GUI state of individual parts of an application's GUI, i.e. without the need of taking the application in the background into account.

3. *Application/system testing:* Testing the collaboration and interactions of all components making up the application.

Fulfiling these goals should adhere to a number of requirements which are directly or indirectly derived from Section **??** and the aforementioned paragraphs. In general, writing tests should be easy and familiar in the sense that (a) learning any involved new tools or API's is not time-consuming, and (b) they consist of mainly standard `C++` and wxWidgets code. Secondly, writing unit tests should be quite similar to writing integration or system tests.

Finally, GUI & application testing specifically demands some additional objectives:

- Don't force the need of learning an extra (i.e. proprietary) scripting language.
- Prevent maintenance problems (cf. [C$^+$04a]) caused by some shortsighted C&R techniques or tools.
- Don't require any internal changes of the employed GUI toolkit, namely wxWidgets, again leading to maintenance problems due to future upgrades for instance.
- It need not be perfect, a pragmatic solution is alright.

1.3 Appyling Unit Testing

The origin of unit testing is without doubt JUnit [Obj04], written by Erich Gamma and Kent Beck initially. Allegedly the initial version was implemented in the airplane on their flight to the OOPSLA'97, a conference devoted OOP, in Atlanta! The best short introduction to the JUnit framework is probably in [C$^+$04b]: "JUnit is a regression testing utility for [Java code ...] intended for white-box testing by the developer, by implementing regression tests in [Java] code. These unit tests are [then] called by JUnit."

Writing the tests in the same language as the IUT has a number of positive effects:

- Both, black-box and white-box tests are very easily implemented.
- Writing the tests is easy and familiar; there is no need to learn a new programming language.

By now there are unit testing frameworks for numerous programming languages around (cf. [C$^+$05, Jef04]), making up the so-called xUnit family. Even the range of of different frameworks for `C++` (cf. [C$^+$04e]), the programming language used by this project, is vast.

This section outlines the decision for a concrete C++ unit testing framework, including reasons and a short introduction on how to use it. Then a typical problem met in applying unit testing, forming a crucial point of any testing efforts with respect to this project, is stated. Finally, several solutions are pointed out in the course of guiding to the "traditional" unit testing solution.

1.3.1 CppUnit

The decision for one unit testing framework from the range of available ones is tough. Each one provides characteristic, distinctive features which are difficult to evaluate without a lot of unit testing experience. Eventually *CppUnit* [Lep05c], originally developed by Feathers, was chosen from the big xUnit family. The main reasons are/were:

- CppUnit is available for a lot of different platforms and compilers (see platform independence requirement in Section **??**).
- It is very similar to JUnit, also with respect to the internals; some people actually denote this as a disadvantage.
- Probably it is one of the most mature frameworks due to its age and active user community.
- The documentation is very illustrative and extensive, making starting with CppUnit easy and fast.
- Last but not least it is open-source software.

A number of steps grounded on automatic generation of code were undertaken to facilitate the application of unit testing with CppUnit on Simpleware projects:

- *Including CppUnit library:* By generating a platform specific CMake usage file in the course of building the SimplewareUtils module (again by means of CMake), the actual inclusion of the CppUnit library to a new testing project requires only some constant CMake project file statements.
- *Test driver:* A template for wxCRP (cf. Section **??**) provides the automatic generation of the complete test driver, see Section A.1.1 for a generation example.
- *Test cases:* Once more a template for wxCRP generates test case class skeletons including a sample test case (if desired).

As shown in Section A.1.2, the generated test case classes implement setUp() and tearDown() methods from the TestFixture class. The reason for the existence of these methods is that usually several little test cases should be executed on a limited set of resources (e.g. database connections), or common objects, called *fixtures*. Sharing common objects can lead to side effects, for instance,

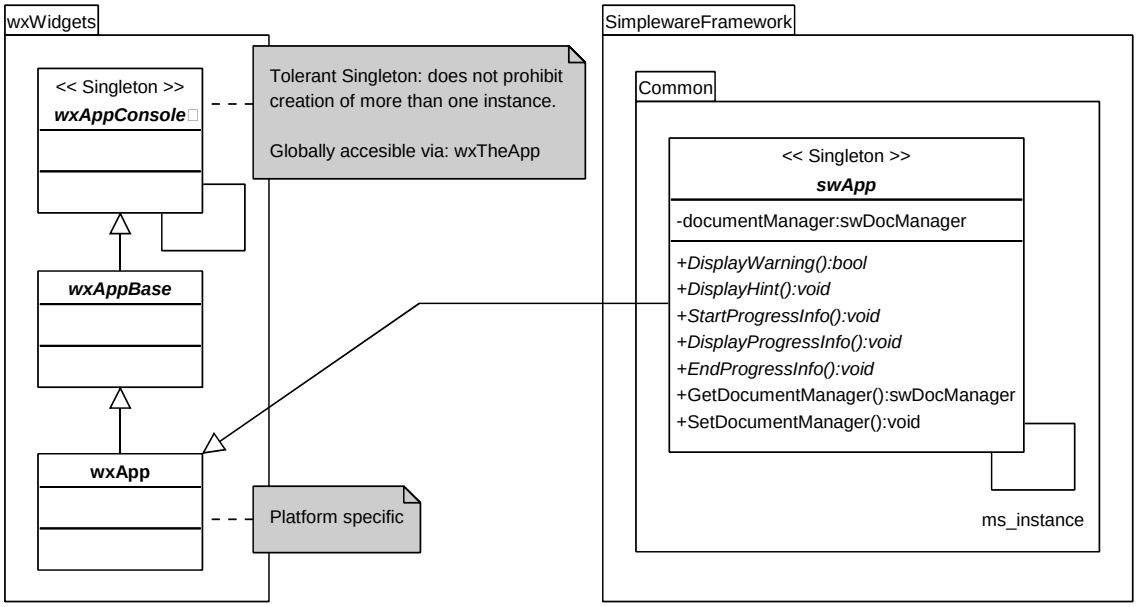


Figure 1.3: Original SimplewareFramework application abstraction.

if one little test case is manipulating a fixture, another little test case's assumption about the state of this fixture will fail. In order to avoid such side effects between the test cases, the setUp() mehod is called every time before a test case is called, allowing the initialisation. Likewise, every time after calling and executing a little test case resources can be freed in the tearDown() method.

1.3.2 Application Dependency Problem

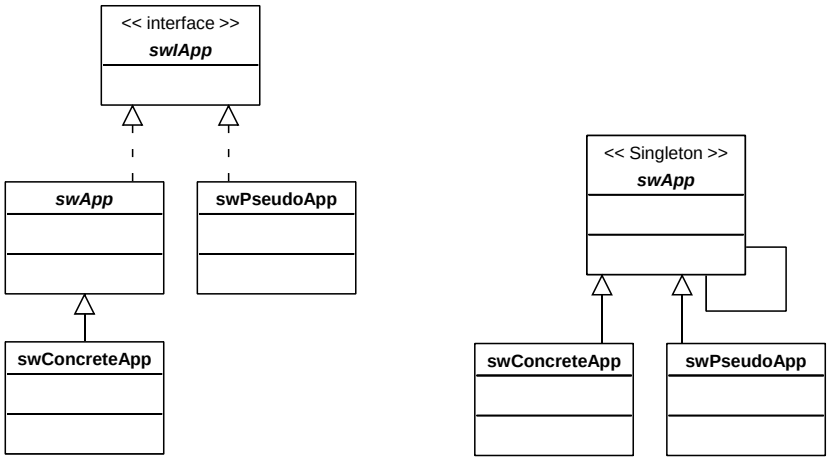
Soon after starting to write unit tests the first hurdle appears: Owing to the intent of developing user-friendly and maintainable applications SimplewareFramework's application abstraction swApp class (Singleton pattern [GHJV94]; see Figure 1.3), which declares methods for displaying errors or progresses of long-lasting operations for instance, is employed. Following the chosen agile development methodology as outlined in Section **??**, code with the need of communicating such information to the user directly uses this abstraction, i.e. there are calls like:

```
swApp::GetInstance()->DisplayWarning(...);
```

The problem is that there is no application instance, let alone a real GUI application instance, because (a) the test driver is the "application", and (b) the GUI must not be accessed or shown. Consequently, without a concrete application instance a high proportion of code cannot be tested.

Adding a further abstraction layer based on the Template pattern [GHJV94] for instance, circumvents this problem:

1. Replacing the call to the single application instance by:



(a) Extracting interface from swApp class.

(b) Extending swApp class.

Figure 1.4: Approaches for swPseudoApp stub design.

```
this->DisplayWarning (...);
```

2. Defining this method as an empty method in the core or business object class:

```
boClass::DisplayWarning (...) { }
```

3. While an application specific sub-class (extending the business object class) eventually delegates it to the single application instance:

```
appClass::DisplayWarning (...) {
  swApp::GetInstance ()->DisplayWarning (...);
}
```

This approach is unacceptable because it leads to twice as many classes and a lot of monotonous coding work which, if done by copy & paste, is error-prone, but can potentially be automated.

Another approach is applying *dependency injection* [Fow04] providing each business object instance with the appropriate application instance. For instance, the real GUI application instance is "injected" to production code, and for testing an application stub implementation is used. Admittedly, dependency injection in combination with an assembler sounds like a good and clean solution, but again adds a lot of extra work to the development: Why not just replace the application singleton instance right away with the application stub implementation? This – using stubs – is the "classical" (unit) testing solution.

Summarising, even testing of methods (the smallest possible code units in OOP) already makes a stub implementation (named swPseudoApp) of the application instance absolutely necessary.

Thinkable stub designs (see Figure 1.4) comprise (a) extracting a so-called swIApp interface from swApp class [Fow00] to be implemented by swPseudoApp class, or (b) inheriting swPseudoApp class from swApp class. Design (a) looks cleaner due to the interface usage, but firstly requires changing the framework's core interface. Admittedly, changing all existing calls of swApp::... to swIApp::... in all existing implementations is a trivial search-and-replace task. Secondly and more important is the fact that future dependencies on wxApp class are not allowed. Unless, swPseudoApp class also extends wxApp class. Finally, the question arises if swApp class should still be the Singleton.

This is even more complex (in the sense of overhead) than stub design (b): Here no changes of existing code or applications are necessary. Moreover, swPseudoApp class instances can potentially have dependencies on wxApp class, whose behaviour can be overridden if desired. Therefore the decision for implementing this approach in SimplewareFramework was made.

As already pointed out, one test class typically has several little test cases in the shape of individual methods (cf. fixtures). In order to allow running each one without making assumptions about their ordering it is necessary to assign an swPseudoApp class instance to the single wxApp instance beforehand. The best way for this is without doubt calling swPseudoApp::SetPseudoAppInstance() method in setUp() method.

The very end of this method (see Listing 1.1) reveals another trap of testing wxWidgets based software: Owing to the heavy usage of static constructs and other little design flaws some framework functionalities (e.g. the colour database, or creation of window objects) require a pre-initialisation before the first usage.

```
void swPseudoApp::SetPseudoAppInstance ()
{
  swApp *app = swApp::GetInstance ();
  if (app != NULL) {
    if (dynamic_cast< swPseudoApp * >(app)) {
      return;
    } else {
      // Should never be reached!
      ::wxLogTrace ("swPseudoApp", "void swPseudoApp::SetPseudoAppInstance ()
              : Instance already set, type is NOT swPseudoApp");
      wxFAIL_MSG ("Instance already set, type is NOT swPseudoApp");
    }
  }

  app = new swPseudoApp ();
  // Initialise wxWidgets application - necessary for global objects like
  // wxTheColourDatabase:
  int argc = 0;
  char **argv = NULL;
  app->Initialize (argc, argv);
}
```

Listing 1.1: Application stub initialisation.

For the sake of implementation facilitation (and error localisation) swPseudoApp class allows to output all information (e.g. warnings, operation progress) on output streams.

Due to applying other SimplewareFramework "patterns", for instance the enhanced MVC pattern and swActiveDocument class, as described in Section **??**, unit tests, as well as non-GUI-integration tests are possible. Section A.1.3 illustrates an example test case for persisting the currently active document in the ScanCAD specific project file format.

1.4 Applying Test-Driven Development

In the course of developing the project TDD was introduced and its application forced. One of the main motivations is the creation of a number of test cases as a by-product. The more successfully running tests, the more confidence in the quality of the code is provided.

For the purpose of investigating the effect of TDD by keeping records of time needs, comments, states/versions of IUT and test cases, a need for tool support arises. A range of development and project management tools are already in use, as mentioned in Section **??**. Utilising some of these tools provides a good starting position due to their familiarity:

- Writing automated tests with the chosen unit testing framework *CppUnit* (cf. Section 1.3.1) is always the first step.
- Recording of time needs and comments is easy with *dotProject*'s task log facility.
- On the other hand, using the Software Configuration Management (SCM) tool *CVS* allows storing versions in addition to time needs and comments.

Consequently, CppUnit and CVS tools are enough for the intended purpose. Redirecting test results from one of various CppUnit test result outputters leads to a high degree of note keeping automation. Unfortunately, using CVS for this purpose must be equated to abusing it and should not be the rule in productive team development: Keeping all intermediate IUT versions and states requires committing the code without having evidence that the code is okay – the test runs have neither been initiated, nor shown to be successful. In fact this is a contradiction to SCM usage in probably any software development methodology. Consequences are of course more dramatic in highly active projects, or for dislocated development teams.

Oftentimes experienced software developers immediately know, or assume, that a certain requirement demands for a solution involving a certain pattern or a combination of design concepts. Thus, in traditional development methodologies they either implement the easy solutions directly, or start designing, for instance using Unified Modelling Language (UML), a solution for the more difficult problems. However, in TDD the procedure asks for breaking down difficult problems into a lot of little rather trivial problems to be implemented according to the rules shortly introduced

in Section 1.1.5; but such real world problems will soon ask for more advise and hints which may or may not be found in Beck's TDD book [Bec02].

The question of how the breakdown should be accomplished, may challenge TDD's core statements and guidelines: should it be based on a preliminary UML design (draft)?

Therefore, pure TDD application examples in this project are mostly on solving small problems which are either small by definition, or are stemming from "difficult" problems allowing a natural or trivial breakdown: functionalities for zipping and unzipping project files, self-cleaning temporary directories, or the registry of expected/provoked warnings for automated application testing. The latter is presented with some detail in Section A.1.4 in the appendix, demonstrating a nice style of recursion: developing parts of the testing framework based on tests. Frankly, the development steps are much bigger than postulated by Beck's TDD recipe. Nevertheless, as depicted by the file time stamps, the time between two steps or commits is a matter of minutes in opposition to XP's demanded daily commit.

1.5 Applying GUI & Application Testing

In contrast to unit testing of (non-GUI) functionalities, GUI and application testing are not straightforward. As pointed out in Section 1.1.6, there are a number of different ways with each one having pros and cons. The innovative, generalisable pragmatic approach to automated testing of GUIs *and* whole applications presented in this work is partly based on ideas of the TFUI mailing list [P+].

While its concrete implementation, a testing framework called *swWxGuiTesting*, is GUI toolkit (i.e. wxWidgets) specific, its underlying ideas are generalisable to most (or all?) other GUI toolkits with only moderate effort; although gaining the same realism and test developer comfort as offered by the presented framework might partly be impossible, and will partly require a major effort. Due to using (a) the cross platform GUI toolkit wxWidgets without internal modifications, and (b) only toolkit specific (non-low-level) concepts available on all platforms supported by the toolkit, not only the testing framework itself is platform independent: swWxGuitTesting allows *platform independent GUI testing*, as well as real TDD of wxWidgets based GUI code and whole applications! (Even many commercial GUI testing tools do not support different platforms; and by nature the tools themselves are rarely platform independent making the maintenance difficult and costly.)

In opposition to application or system testing, GUI testing means testing parts of an application's GUI (or GUI parts of an application), but without the need of taking the application itself into account. Nevertheless, the method to replace the real GUI application instance with a stub as presented in Section 1.3.2 is still valid, thus, permitting the testing of GUI code with dependencies on an application instance.

This section continues with presenting underlying concepts for GUI testing before combining them

into a core solution. Then this solution is extended to support basic application testing. Subsequently, potential problems (ignored for basic application testing) and their solutions are demonstrated. Finally, the actual implementation of GUI test cases including required changes of future SUTs and facilitating helper classes is shown, before concluding with adding C&R functionality to this automated GUI and application testing approach.

1.5.1 Underlying Concepts

In unit testing the test driver is iterating over all known test cases to run them consecutively. Thus, test cases, or more precisely the test routines made up of a number of test steps or instructions, control the code flow.

By contrast, GUI based code or applications are event driven, i.e. GUI events initiated by user input control the code flow:

1. GUI is created.
2. User is interacting with the GUI ⇒ GUI events are generated.
3. These events are processed and corresponding callback handler methods are executed.

Solving this conflict between unit testing and GUIs with respect to code flow controlling and event simulation necessitates answering the following questions:

- Which system is starting and controlling the other system?
- How to avoid loosing control over the other system when starting it?
- How to simulate GUI events?
- How to keep or regain control after simulating GUI events in test routines?
- Are there any thread related issues?

The following sections will present ideas and solutions to all these questions before combining them to the GUI testing core solution in the next section.

1.5.1.1 Which system is starting and controlling the other system?

In general, there are two possible answers:

- The CppUnit test driver creates the GUI (GUI testing), or starts the application (application testing).
- The application starts the CppUnit test driver; but what about mere GUI testing?

A shortsighted idea for implementing the second procedure could be linking test routines with specifically added GUI controls. That is, adding a menu item for each test case which will run this test case at menu item selection for instance. Another way could be starting the test driver right at the end of the application's initialisation phase. None of these ideas is acceptable due to a combination of the following reasons: (a) partly heavy modifications of GUI or application are necessary (for each test case); (b) some GUI parts under test do not support menus; (c) the activation of test cases via the menu is not automated; (d) testing GUIs and GUI callbacks is problematic from another GUI callback; (e) GUI testing requires a special GUI application dummy implementation.

Therefore procedure one, creating the GUI or application from the test driver, requiring the inversion of the normal GUI controlled code flow seems to be inevitable for GUI (unit) testing.

In a first naive approach wxTestRunner [PL05], a GUI test driver for CppUnit using wxWidgets, sounds like a good solution to this problem. Owing to its GUI interface a fully initialised wxWidgets application is ready for running GUI test cases. However, it also poses a huge problem: Being a GUI application, wxTestRunner replaces the single GUI application instance, and so rules out application testing. A feasible method to mitigate this problem is implementing as much functionality as possible outside the application class. For instance, applying a "move out" refactoring [Fow00] on the main frame will allow testing the main frame as well. Nevertheless, some application aspects are virtually impossible to test.

A better approach involves firstly, initialising wxWidgets library from the command line based test driver, and secondly, instantiating the SUT (or a stub application for GUI testing). Because wxWidgets applications can be command line tools as well, different ways of initialisation are provided. The rough idea of simulating the GUI without ever showing it, and thus, only initialising the so-called console mode by means of ::wxInitialize() function fails: any GUI "simulation interaction" requires sending messages or posting events, but in view of the missing main (message) loop these messages are floating the system and result in a stack overflow. Whereas adding this main loop by copying from the standard GUI application class wxApp sounds naturally, it poses other questions: (a) is this really enough to simulate GUIs, and (b) why not directly initialise the GUI mode and use this class' code instead of copying it?

In contrast to the console mode of wxWidgets, the GUI mode can only be initialised and destructed once in the lifetime of a process due to some design flaws, for instance the heavy usage of static constructs. The documentation of ::wxEntry() function performing this initialisation says: "This initializes wxWindows in a platform-dependent way. Use this if you are not using the default wxWindows entry code (e.g. [...] WinMain [for Microsoft Windows GUI applications]). For example, you can initialize wxWindows from an Microsoft Foundation Classes application using this function." [wxW05b] Potential solutions to this only-once-in-a-lifetime initialisation problem involve moving all test cases into a separate library and running each one in its own process.

1.5.1.2 How to avoid loosing control over the other system when starting it?

The previous section revealed that starting the SUT from the CppUnit test driver is by far the better option. Unfortunately the default implementation of the initialisation function results in the test driver's control loss, thus, the GUI is processing normally and the test driver will only regain control after the GUI is quit and destructed.

The solution to this problem is applying the "Then Don't Call Main Loop" pattern [C+04d], but in a different context (cf. Section 1.1.6) and in anticipation of quite different intents.

Extremely simplified the GUI mode initialisation with respect to the concrete application life cycle consists of the following steps:

1. Query existing application instance myApp (or create a dummy console application)
2. Initialise application (myApp−>OnInit())
3. Run application (wxAppBase::OnRun())
4. Which in turn executes the main loop (wxApp::MainLoop())
5. Exit application (myApp−>OnExit())

This is also depicted in the sequence diagram in Figure 1.5, where some UML "tricks" are necessary for underlining: (a) the function stereotype is a common solution for modelling functional code with UML, and (b) it is tried to emphasise the inheritance hierarchy by distinguishing the methods according to their actual implementation classes (which are ordered by inheritance from left to right).

Thus, the crux of the problem is the main loop execution initiated by wxAppBase::OnRun() method. By overriding this method so that it does not call wxApp::MainLoop() method, the code flow control is not "handed over" or lost to the GUI forever. Overriding MainLoop() method instead is not an option because of the main loop's dramatic platform specific differences. Furthermore, this – overridden OnRun() method – is also the perfect place for running test cases if they are known, and accessible...

1.5.1.3 How to simulate GUI events?

Testing always aims at testing as much involved code as possible. Hence, instead of, for instance, calling a callback handler method directly, simulating the corresponding event in the test routine is the ultimate goal.

wxWidgets represents events in the shape of wxEvent class subclasses. Zeitlin, one of wxWidgets' core developers, states: "Basic[al]ly, creating wxEvents is very easy – but you can't simulate the low[-]level events from which the wxEvents are normally generated." [Zei03] That is, the wxWidgets

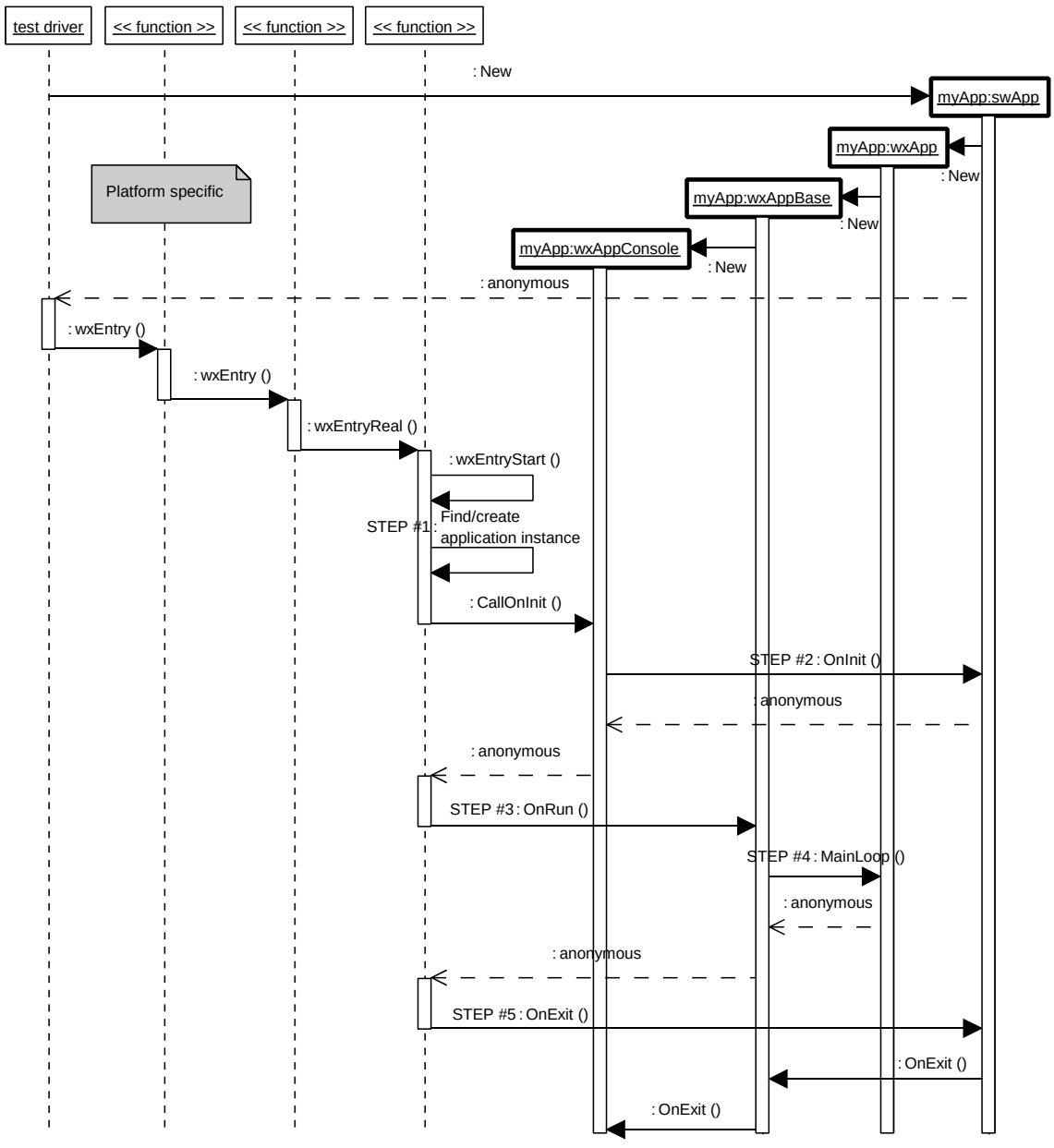


Figure 1.5: Sequence diagram: Initialisation of wxWidgets (application).

event system is processing platform specific low-level events and converts them into (platform independent) wxEvent instances for further processing. Thus, the rest of the framework can be independent of platform specific event issues. Consequently, the simulation of pressing buttons requires special button click events; sending either mouse button down events, or enter key events to wxButtons fails.

By contrast, the Java GUI testing framework jfcUnit simulates events on a lower level using AWT. Here simulating mouse clicks to press buttons is the most straightforward way. Helper methods like enterClickAndLeave() method make writing realistic tests easy. By default, this implementation will perform the following steps for a button target:

1. Calculate the button's centre location.
2. Move the mouse pointer in configurable steps to the button's centre.
3. Generate a low-level mouse click event with the button's centre as x/y coordinates.

To recapitulate simulating events in wxWidgets consists of (1) creating high-level events, and (2) sending or posting these events. While step one means filling event containers with at least the ID of the affected control, but oftentimes additional event specific (partly duplicate) information, step two only involves handing over the created events to corresponding event handlers. In most cases this is the affected control's parent control. In contrast to sending events by means of wxEvtHandler::ProcessEvent() method, where the event is processed immediately before the function returns, posting events (wxEvtHandler::AddPendingEvent() method or ::wxPostEvent() function) is the recommended thread-safe way to enqueue another event in the event queue to be processed in a later event loop iteration. Unfortunately, there are some event specific differences demanding event sending in some cases, e.g. notebook page selection. See also Listing 1.2 for the probably most trivial event simulation example: selecting a menu event.

1.5.1.4 How to keep or regain control after simulating GUI events in test routines?

As outlined in the previous section test routines simulate GUI events to increase testing code coverage. Test routines usually contain a number of steps and require simulating several events. Therefore "single stepping" (in analogy to debugging) through the current test routine must continue after each event simulation.

A first idea involves employing a special reserved event similar to the wxTest project [Sax01a], another xUnit testing framework introducing an "interface specification between tests and test-runners [...facilitating a] complete binary decoupling" [Sax01b]. Unfortunately, this project is neither platform independent, nor mature due to being inactive since version 0.2.

This special event serves the purpose of communicating the end of event simulations. It is processed by special event handlers based on wxWidgets' event handling system which allows pushing event

handlers on top of the current event handler stack. One disadvantages of this idea is the highly likelihood that another piece of code is pushing another event handler on top of this one. So, modifying wxWidgets in order to allow event handler stack change notifications and stack re-orderings is necessary.

The aforementioned Java GUI testing framework jfcUnit provides another solution to this problem by allowing flushing the event queue after posting events. I.e. all events in the queue are processed before the thread control is returning. One way of applying this approach on wxWidgets based applications without modifying wxWidgets, or overriding the message processing loop rests upon the concept of idle time. As stated in [wxW05c] "idle events [...] are generated when the system is idle", and "idle time [...is] when all messages have been processed" [wxW05a]. Thus, typical GUI test case core constructs resemble Listing 1.2: (1) create event, (2) post event, and (3) flush event queue.

```
...
frame->show (true);
FlushEventQueue ();
// Simulate menu selection:
wxCommandEvent menuEvt (wxEVT_COMMAND_MENU_SELECTED, THE_MENU_ITEM_ID);
::wxPostEvent (frame, menuEvt);
FlushEventQueue ();
...
```

Listing 1.2: Example GUI test case excerpt.

Once again, thanks to the fact that wxWidgets is open-source and its source code is therefore freely available, a clean way for detecting idle time is possible via overriding the undocumented internal wxApp::OnIdle() method. This method can be registered to get notified at/of idle time. FlushEventQueue() method (as used in Listing 1.2) is responsible for processing all events currently in the event queue and must return at idle time. The solution for this lies in the trick of exiting the main loop in OnIdle() method: Listing 1.3, depicting FlushEventQueue() method, shows setting a flag indicating to exit on idle time detection, before executing the main loop. This flag is then queried in the overridden OnIdle() method to conditionally exit the main loop in turn, giving back the control to the caller of FlushEventQueue() method (see Listing 1.4).

```
void FlushEventQueue () {
  SetExitMainLoopOnIdle (true);
  // wxTheApp = single wxWidgets application instance
  wxTheApp->MainLoop ();
}
```

Listing 1.3: Flushing event queue.

The abuse of idle events is the most likely problem of this solution. That is, it is important to find out where idle events are generated because these event generation sources can potentially

```
void MyApp::OnIdle (wxIdleEvent &event) {
  wxApp::OnIdle (event);
  if (IsExitMainLoopOnIdle ()) {
    this->ExitMainLoop ();
  }
}
```

Listing 1.4: Exiting main loop on idle time detection.

interfere with the event queue flushing which stops in OnIdle() method. Fortunately, there are only two groups of idle event sources, and none of them is problematic: Group one is application specific and thus, their use (or abuse) is solely up to the developer (wxApp::SendIdleEvents(), wxApp::ProcessIdle() and wxApp::WakeUpIdle() methods). Group two only deals with event processing (or thread communication via events) (::PostEvent() function and wxEvtHandler::AddPendingEvent() method).

The idea of gaining independence from OnIdle() method usage by going one level deeper is doomed to fail. Recognising idle time earlier, i.e. detecting it in the main loop preventing the need to call OnIdle() method, and exiting the main loop immediately, poses two major problems: (a) no platform independency, and (b) again a maintenance nightmare because wxWidgets code is likely to demand modifications.

1.5.1.5 Are there any thread related issues?

In contrast to, for example, Java GUI applications based on AWT or Swing, wxWidgets does not create an extra GUI thread in addition to the main thread. While it is recommended that only the main thread should interact with the GUI, this is not explicitly programmatically restricted. Having said that, only the main thread is allowed to execute some code bits:

- ::wxMutexGuiLeaveOrEnter() function (only non-main threads are allowed to call ::wxMutexGuiEnter() and ::wxMutexGuiLeave() functions on the other hand)
- wxThread::SetConcurrency() method
- ::wxExecute() function
- wxApp::WakeUpIdle() method (use ::wxWakeUpIdle() function from any thread)
- wxApp::Yield() method (use wxThread::Yield() function from any thread)
- wxTimerBase::Start() method
- wxEvtHandler::ProcessEvent() (use wxEvtHandler::ProcessThreadEvent() from any thread)

Also, the main thread is rigorously responsible for the message/event handling and processing. Again, moving test cases into a separate library and process, as stated with respect to wxWidgets' GUI application mode specific only-once-in-a-lifetime initialisation problem, potentially circumvents this thread related issue if necessary.

As mentioned in the previous section, jfcUnit allows flushing the event queue, that is, processing or handling of all posted simulation events. Due to the inherent threading issue (cf. extra GUI thread creation) the involved steps always comprise (1) synchronising the GUI access by pausing the main thread (pauseAWT() method), and (2) processing (simulated) events (flushAWT() method). In analogy wxWidgets would provide ::wxMutexGuiLeaveOrEnter(), ::wxMutexGuiEnter() and ::wxMutexGuiLeave() functions for the first step, but in normal (or most) cases the threading issue is not applicable.

In conclusion, the advantage of putting GUI applications into service with no other than the main thread leads to easier coding in most cases. It avoids the need for taking multithreading issues like synchronisation of access to shared objects into account. Finally, it also strengthens the case against employing the aforementioned wxTestRunner as code flow controller: wxTestRunner is not a general solution because the main thread is running the wxTestRunner appliction, whereas test cases are run in child threads.

1.5.2 GUI Testing Core Solution

After presenting the underlying concepts, this section describes the core solution of automated GUI testing in detail. Exceptional cases like modal dialog blockings are ignored and dealt with in subsequent sections. To recapitulate, the required steps are:

- Controlling the SUT from the test driver.
- Initialising the application in wxWidgets GUI mode exactly once before running the first GUI test case.
- Not starting the main loop to allow "single stepping" through test routines.
- Simulating events based on event queue flushing by temporarily running main loop and exiting the main loop on idle time.

The underlying concepts where applied directly on the SUT for the sake of facility, but this is not acceptable as it would require changing each SUT depending on testing vs. running mode. So, the remaining open issues are:

- How to add GUI test cases in comparision with "normal" test cases?
- How to concretely integrate wxWidgets (application) with CppUnit (test driver): one-time initialisation of GUI mode and code flow control splitting/diversion?

- How to minimise modifications required to make an application testable (i.e. turning it into an SUT)?
- How to allow GUI *and* application testing?

In order to run GUI test cases after GUI mode initialisation it is necessary to clearly identify them. Therefore a separate (registry) test case suite named "WxGuiTest" must contain all GUI test cases (side note: a test suite is again a test case permitting hierarchical structuring; cf. Composite Pattern [GHJV94]). Adding them is following the advise in [FL], thus, employing the TestFactoryRegistry class which allows registering test suites at initialisation time and solves two pitfalls: (a) forgetting to add test cases to the test runner, and (b) compilation bottleneck caused by the inclusion of all test case headers. While adding test cases to test suites is identical for GUI and non-GUI test cases (see Listing 1.5), registering the respective test suites differs only in a single line. Again, templates for wxCRP (cf. Section **??**) provide the automatic generation of test case skeletons including this little difference. Non-Gui test suites are registered by:

```
// Register test suite:
CPPUNIT_TEST_SUITE_REGISTRATION( swSIPFileVersionFactoryTest );
```

, whereas GUI test suites must use:

```
// Register test suite with special name in order to be identifiable as test
// which must be run after wxWidgets GUI mode is initialised:
CPPUNIT_TEST_SUITE_NAMED_REGISTRATION( ExampleWxGuiTest, "WxGuiTest" );
```

```
class swSIPFileVersionFactoryTest : public CPPUNIT_NS::TestFixture
{
  CPPUNIT_TEST_SUITE( swSIPFileVersionFactoryTest );
    CPPUNIT_TEST( testCreateSIPv20 );
    CPPUNIT_TEST( testGetLatestVersion );
  CPPUNIT_TEST_SUITE_END();
...
```

Listing 1.5: Adding test cases to test suites.

Integrating wxWidgets (application) with CppUnit (test driver) with respect to one-time initialisation of wxWidgets GUI mode employs the Decorator pattern [GHJV94]. The inspiration for this solution comes from JUnit (cf. [Bee04] and [Cla04]). Even the involved classes have more or less the same names demonstrating the amazing internal similarities between JUnit and CppUnit. Figure 1.6 depicts the structure of the involved classes in the class diagram.

As shown in Listing 1.6 swInitWxGuiTest class creates the decoration of all registered GUI test cases by means of the static suite() method. This method is handing over all registered GUI test cases to a new swInitWxGuiTestSetUp class instance, which in turn does the actual decoration. By finally returning this instance the decoration is transparent to CppUnit.

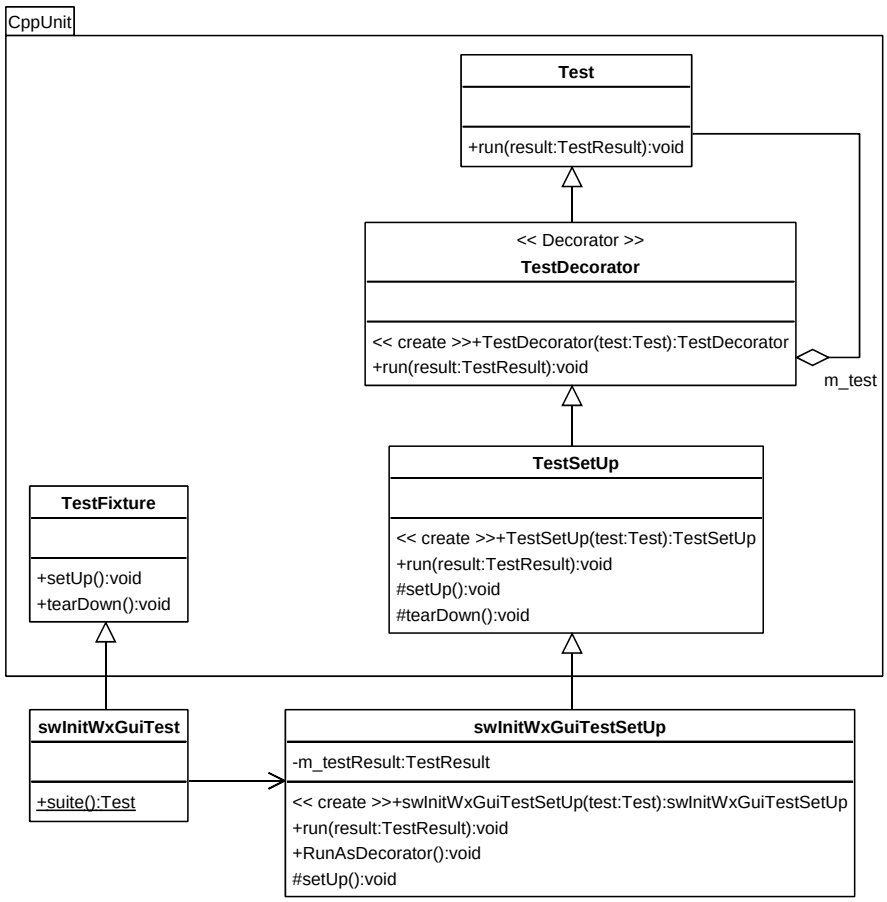


Figure 1.6: Class diagram: One-time initialisation of wxWidgets GUI mode with Decorator pattern.

The usage from GUI test drivers is slightly more complicated than from normal test drivers because firstly, swInitWxGuiTest class must not be added to the test registry in the swWxGuiTesting library, but in the actual testing project; and secondly, for the sake of generality a new test suite equally named to the usual top level registry suite is created for holding the decorated test cases. Once more this difference requires not more than selecting the appropriate wxCRP template (generating the whole test driver) by developers. So instead of:

```
int main (int argc, char* argv[]) {
  // Get the top level suite from the registry
  CPPUNIT_NS::Test *suite = CPPUNIT_NS::TestFactoryRegistry::getRegistry().
          makeTest();
...
```

GUI test driver's code begins with:

```
int main (int argc, char* argv[]) {
  CPPUNIT_NS::TestSuite *suite = new CPPUNIT_NS::TestSuite ("All Tests");
  suite->addTest (swTst::swInitWxGuiTest::suite ());
...
```

```
CPPUNIT_NS::Test * swInitWxGuiTest::suite () {
  CPPUNIT_NS::TestSuite *suiteOfTests = new CPPUNIT_NS::TestSuite ();

  // Add all tests of specially named registry as well:
  CPPUNIT_NS::Test *wxGuiTestSuite = CPPUNIT_NS::TestFactoryRegistry::
      getRegistry ("WxGuiTest").makeTest ();
  suiteOfTests->addTest (wxGuiTestSuite);

  swTst::swInitWxGuiTestSetUp *initWxGuiTestSetUp =
      new swTst::swInitWxGuiTestSetUp (suiteOfTests);

  return initWxGuiTestSetUp;
}
```

Listing 1.6: Creating decorated GUI test cases.

Finally, Listing 1.7 reveals the one-time initialisation of wxWidgets GUI mode in swInitWxGuiTestSetUp::setUp() method, as well as the required code flow diversion with respect to running the decorated GUI test cases. This code flow diversion forms the important second wxWidgets-CppUnit integration step as explained in the following.

```
void swInitWxGuiTestSetUp::run (CPPUNIT_NS::TestResult *result) {
  // Store result for latter call of TestDecorator::run(result) to run all
  // test cases:
  m_result = result;

  this->setUp ();
  // Do NOT call because flow of control must be diverted:
  //TestDecorator::run (result);
  this->tearDown ();
}


void swInitWxGuiTestSetUp::RunAsDecorator () {
  TestDecorator::run (m_result);
}


void swInitWxGuiTestSetUp::setUp () {
  // Do the one-time initialisation of wxWidgets GUI mode:
  ...
}
```

Listing 1.7: Adding test cases to test suites.

For the purpose of (a) minimising modifications of each application turning it into an SUT, and (b) allowing mere GUI testing, a kind of application instance stub in the shape of Decorator and Proxy pattern [GHJV94] is used (this becomes clearer in the subsequent section about application testing). This application instance of swWxGuiTestApp class (see Figure 1.7 for the class diagram), is inheriting from swPseudoApp class (cf. application dependency problems in Section 1.3.2) and

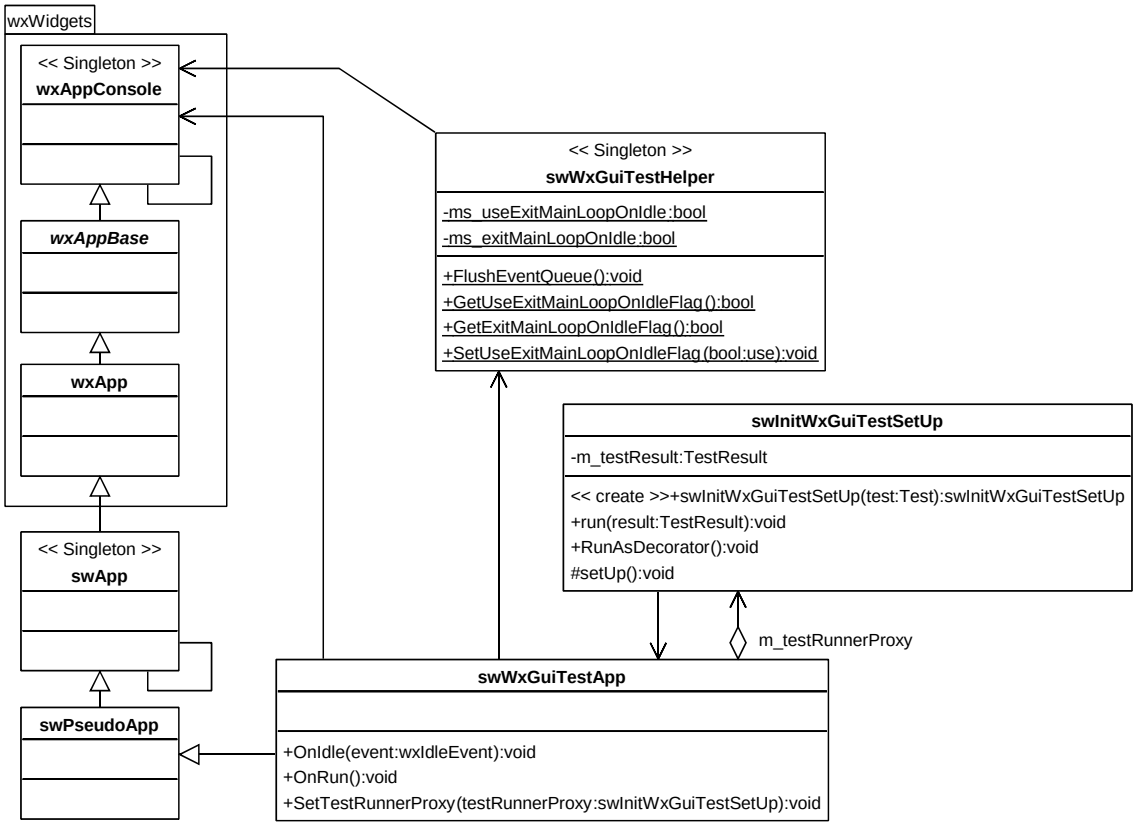


Figure 1.7: Class diagram: swWxGuiTestApp class with interactions for GUI testing.

is used as the singleton instance from wxApp* class hierarchy. This is also a reason why another Singleton pattern in swApp class is mandatory: otherwise there cannot be an SUT (for application testing) besides the GUI testing specific swWxGuiTestApp class, requiring to put all its functionality into the SUT.

As already indicated, swInitWxGuiTestSetUp::setUp() method (see Listing 1.8) performs the one-time initialisation of wxWidgets GUI mode. Additionally, it also instantiates the swWxGuiTestApp class and applies setter dependency injection with itself on this instance. Thus, instead of CppUnit's test driver, swWxGuiTestApp instance controls the code flow by running the decorated test cases from swWxGuiTestApp::OnRun() method (see Listing 1.9). The IMPLEMENT_APP_NO_MAIN(...) macro at the top of the implementation of this class is crucial in connection with the one-time initialisation via ::wxEntry() function. This class also contains the bulk of the "single stepping" implementation as previously described, thus, minimising SUT modification needs. Only flushing the event queue is implemented by swWxGuiTestHelper::FlushEventQueue() method in analogy to Listing 1.3.

A simplified sequence diagram (see Figure A.1) in Section A.1.5 in the appendix shows the start of running test cases, beginning from test driver up to solely one rather symbolic event simulation within the test case. Although the diagram omits the initialisation of wxWidgets wherever possible,

```
void swInitWxGuiTestSetUp::setUp () {
  swWxGuiTestApp *wxGuiTestApp = new swWxGuiTestApp ();
  // This is not really necessary, as done automatically by code from the
  // macro expansion; but it improves understanding:
  wxApp::SetInstance (wxGuiTestApp);

  // Store this instance for running tests from swWxGuiTestApp::OnRun():
  wxGuiTestApp->SetTestRunnerProxy (*this);

  #if defined (WIN32)
    #if !defined (__BUILTIN__)
      ::wxEntry (GetModuleHandle (NULL), NULL, NULL, SW_SHOWNORMAL);
    #else
      ::wxEntry (GetModuleHandle (NULL), NULL, NULL, SW_SHOW);
    #endif
  #else
    ::wxEntry (i_args, p_args);
  #endif
}
```

Listing 1.8: wxWidgets-CppUnit integration - one-time initialisation of wxWidgets GUI mode.

or simplifies it in the remaining cases, the complexity behind this testing framework becomes visible. While this diagram is not showing GUI testing, but application testing, this does not add any complexity to the diagram with the little exception of the OnInit() method delegation. This, and the fact that employing swWxGuiTesting is comparatively trivial (cf. Section 1.5.5) demonstrates the elegance of the developed solution. For the sake of clarity the user defined test case including the event simulation is singled out and depticted in Figure 1.8.

This core solution supports GUI testing but does not take application testing explicitly into account. Extending this solution to support application testing as well is described in the next section.

1.5.3 Application Testing

For testing whole applications the swWxGuiTesting framework has to be extended because it already contains a wxApp* class singleton, namely swWxGuiTestApp class. This is closely linked with wxWidgets' application initialisation based on IMPLEMENT_APP*(...) macros: As already pointed out the swWxGuiTestApp instance serves as an application stub in GUI testing being initialised by means of ::wxEntry() function and IMPLEMENT_APP_NO_MAIN(swWxGuiTestApp) macro. Normal applications employ IMPLEMENT_APP(sccScanCADApp) macro instead.

Moving the solely testing related code from swWxGuiTestApp class into each application, turning it into an SUT, is a maintenance nightmare. While minimising required modifications of applications to turn them into SUTs is one side, making the change between normal application running and application testing just a matter of switching is the other side.

Based on the build tool CMake (cf. Section **??**) and a conditional preprocessor macro redefinition

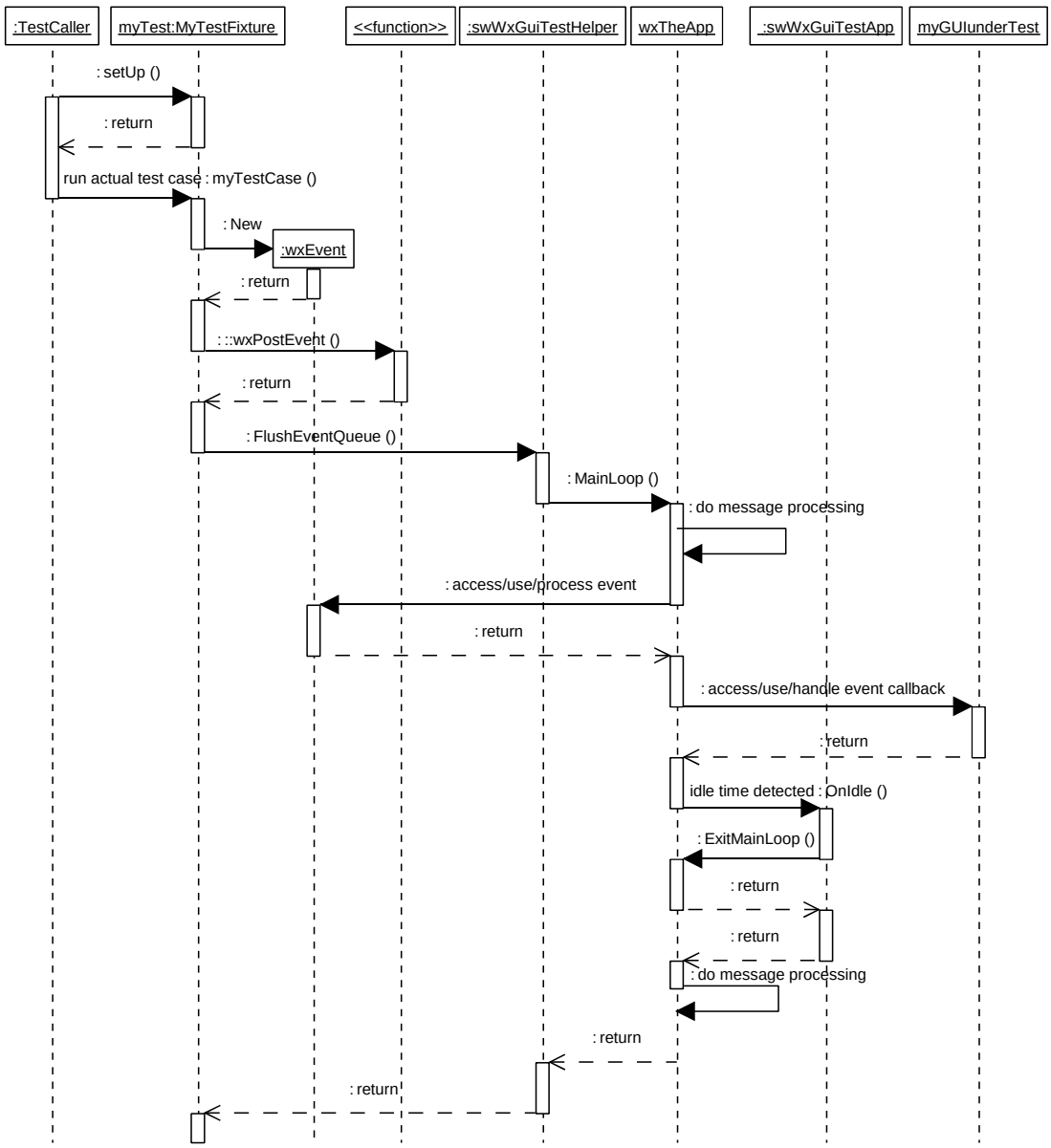


Figure 1.8: Sequence diagram: Test case event simulation.

```
IMPLEMENT_APP_NO_MAIN(swWxGuiTestApp)

...

void swWxGuiTestApp::OnIdle (wxIdleEvent &event) {
  swPseudoApp::OnIdle (event);

  if (swWxGuiTestHelper::GetUseExitMainLoopOnIdleFlag () &&
      swWxGuiTestHelper::GetExitMainLoopOnIdleFlag ()) {
    wxTheApp->ExitMainLoop ();
  }
}


int swWxGuiTestApp::OnRun () {
  ...

  wxASSERT (m_testRunnerProxy);
  m_testRunnerProxy->RunAsDecorator ();

  // For a clean exit all windows must be closed; otherwise their "proxy"
  // classes cannot be de-registered. At least the existence of the
  // applications top window can be checked & closed by default:
  if (wxTheApp->GetTopWindow ()) {
    wxTheApp->GetTopWindow ()->Close ();
  }
  // Finally process all pending events:
  swWxGuiTestHelper::SetUseExitMainLoopOnIdleFlag (false);
  return wxTheApp->MainLoop ();
}
```

Listing 1.9: wxWidgets-CppUnit integration - code flow control diversion in swWxGuiTestApp class.

the necessary changes for an application (= future SUT) are easily managed:

- A separate (project) build target deals with the macro usage in the application's implementation file. This target must contain the application class files and a special, reserved definition; the corresponding "CMakeLists.txt" contains entries like:

```
ADD_EXECUTABLE (CppWxAppGuiTesting
  ${APP_SRCS}
  ${CPPUNIT_TEST_SRCS}
)

ADD_DEFINITIONS (-DSW_USE_WX_APP_GUI_TESTING)
```

- The concrete macro redefinition is moved into the swWxGuiTestHelper class header file from swWxGuiTesting framework in order to follow the Don't Repeat Yourself (DRY) principle [HT99]. It redefines the macro to expand to nothing when the build target holds the aforementioned special definition:

```
#ifdef SW_USE_WX_APP_GUI_TESTING
  #ifdef IMPLEMENT_APP
    #undef IMPLEMENT_APP
  #endif
  #define IMPLEMENT_APP(appname)
#endif
```

- Finally, the application's implementation class must only include this swWxGuiTestHelper header file which also provides other important functionalities required for testing in any case.

With the aforementioned mainly maintenance related ideas in mind, neither subclassing the application, nor subclassing swWxGuiTestApp class is satisfactory. The former is possibly the most naive maintenance problem cause: adding code from swWxGuiTestApp class is trivial, but must be done for each application in a copy & paste manner. The latter, subclassing swWxGuiTestApp class (⇒ new swWxAppGuiTestApp class) leads again to singleton instance problems caused by the macro based application instantiation.

Extending swWxGuiTestApp class to combine GUI and application testing, plus using delegation from swWxGuiTestApp to the SUT is a clean solution. Thus, this class is a mixture of Decorator and Proxy pattern, as indicated in Section 1.5.2. This solution also makes the eventual usage easier, because there is only one framework for both purposes; and using one or the other is straightforward as presented in the following. The extension of GUI testing allowing application testing requires the following modifications (see Listing 1.10 for swInitWxGuiTestSetUp::setUp() method and Listing 1.11 for swWxGuiTestApp class):

- In order to allow correctly initialising and setting up the involved classes and objects, the test driver first instantiates the actual SUT. Afterwards the single swApp instance points to the SUT:

```
int main (int argc, char* argv[]) {
  ...
  // Create SUT:
  scc::sccScanCADApp *scanCADApp = new scc::sccScanCADApp();
  ...
```

- By temporarily storing single swApp instance, instantiating the swWxGuiTestApp class prevents loosing the SUT object in swInitWxGuiTestSetUp::setUp() method.
- In comparison to mere GUI testing this swWxGuiTestApp class instantiation employs constructor dependency injection with the SUT object, giving the swWxGuiTestApp instance access to the SUT.
- After creating all objects the singleton instances have to be corrected: wxApp singleton must point to swWxGuiTestApp instance, whereas swApp singleton must point to the SUT object.

- Finally, wxApp class methods of relevance to the SUT are overridden in swWxGuiTestApp class for the purpose of delegating them to the SUT object. Usually OnInit() and OnExit() methods are enough. Methods from swApp are still directly called on SUT because the SUT object is the single swApp instance.

```
void swInitWxGuiTestSetUp::setUp () {
  // There is already an swApp instance which is supposed to be the real
  // application under test. Otherwise "only" wxGuiTesting is carried out:
  swApp *app = swApp::GetInstance ();
  if (app != NULL) {
    swApp::Nullify ();
  }

  swWxGuiTestApp *wxGuiTestApp = new swWxGuiTestApp (app);
  // This is not really necessary, as done automatically by code from the
  // macro expansion; but it improves understanding:
  wxApp::SetInstance (wxGuiTestApp);
  // And re-establish SUT as single swApp instance (if it is not just a GUI
        test):
  if (app != NULL) {
    swApp::SetInstance (app);
  }
  ...
```

Listing 1.10: swInitWxGuiTestSetUp::setUp() method allowing application testing in addition to GUI testing.

The call of swApp::Nullify() method in swInitWxGuiTestSetUp::setUp() method (as shown in Listing 1.10) is only necessary due to defensive programming. swApp class constructor and swApp::SetInstance() method assert that no instance has already been created before proceeding. Thus, by setting the instance to NULL it is pretended that this condition is fulfilled.

Finally, as already indicated in GUI testing's core solution, Section A.1.5 in the appendix shows a simplified sequence diagram (see Figure A.1) of a symbolic test case from test driver start to one event simulation within the test case. The complexity of integrating wxWindows with CppUnit and the required code flow splitting between these two systems becomes obvious. This and the fact that GUI and application testing share the bulk of implementation code, as well as eventual usage, demonstrates the developed solution's elegance.

1.5.4 Blocking Problems

Applications, and therefore SUTs, oftentimes use code constructs interfering with the main loop (i.e. they have their own message loop), or requiring feedback from the user. This results in halting or pausing test runs necessitating manual user input, like pressing the Enter key for instance, to resume. Definitely this contradicts the aim of automated testing. Similarly, GUI testing is likely

```
swWxGuiTestApp::swWxGuiTestApp (swApp *appUnderTest) {
  m_testRunnerProxy = NULL;
  m_appUnderTest = appUnderTest;
}

...

// Methods overridden for delegation to SUT:
bool swWxGuiTestApp::OnInit () {
  if (m_appUnderTest != NULL) {
    // Somehow in release mode argc was initialised to an arbitrary value
    // making SUT->OnInit() thinking there is a document specified on the
    // command line to load up:
    m_appUnderTest->argc = 0;
    return m_appUnderTest->OnInit ();
  } else {
    return swPseudoApp::OnInit ();
  }
}


int swWxGuiTestApp::OnExit () {
  ...
}
```

Listing 1.11: swWxGuiTestApp class allowing application testing in addition to GUI testing.

to be exposed exactly the same problems as application testing, as soon as the testing coverage increases and a certain integration testing level is reached.

Code constructs causing test blocking problems are:

- Modal dialogs
- Message boxes
- Pop-up menus
- Failing assertions in debug mode
- Unhandled exceptions

Thus, detecting or avoiding these constructs, or manipulating their effects which are outlined in this paragraph is vital. Because wxWidgets builds message boxes out of modal dialogs on all supported platforms, they share the standard behaviour of running in their own "temporary" message loop waiting for user input. Moreover, at least on the Microsoft Windows platform they do not send any idle events. Likewise, pop-up menus only return control after dismission by the user. By default failing assertions pop up modal dialogs, asking the user whether she wants to abort the program, continue or continue ignoring any subsequent assert failures. Finally, unhandled

exceptions should not occur due to defensive programming by adding try-catch blocks around each potentially exception throwing piece of code; but if they occur, the application will usually crash.

Taking the requirement of not changing wxWidgets into account it is impossible to avoid all occurrences of these code constructs: some are certainly inherent in the GUI toolkit itself. For instance, standard dialogs (e.g. for opening or printing files) are modal under most (or all?) platforms.

Modifying wxWidgets is necessary to solve these problems for all platforms within the toolkit, but this is a maintenance nightmare considering future upgrades of wxWidgets versions. Even using, for instance, Perl scripts allowing extremely powerful text processing and regular expressions is no option. The difficulty of parsing C++ grammar is well-known and would require reinventing the wheel which is probable doomed to fail anyway [Bax03]. On the other hand Aspect-Oriented Programming (AOP) [AOS] could manage this modification task without maintenance problems. Owing to AOPs inherent nature each type of modification can be a separate cross cutting concern (CCC) and therefore a separate stand-alone aspect. Unfortunately, the available AOP tools for C++, namely Aspect++ [pur05] and XWeaver [Pas05], were deemed too immature and unreliable. (This in fact led to the aforementioned "non-modification" requirement.)

On a second thought all testing stoppers have one thing in common. Sooner or later after their occurrence the window, dialog or application focus changes. That is, another part of the GUI becomes active. While wxWidgets seems to provide appropriate functionality to detect such de-/activations by means of activate and focus events, a function to query the active window, dialog initialisation events, and closing events, this is not at all sufficient. So for instance:

- the activation event handler only detects deactivations of main (top-level) windows, but not the activation or return unless the event handler is pushed on the application's top-level window; activation changes between non-top-level windows do not lead to any events.
- the application activation event is only working under Microsoft Windows platform.
- ::wxGetActiveWindow() function supposedly also works only under Microsoft Windows, but in fact not even there.

So, as it turns out, detecting the blockers is not possible (in general), asking for ways to avoid them. Again, toolkit inherent blockers can only be avoided by not accessing any toolkit functionality depending on code using these code constructs. In some cases (e.g. code using standard file open dialogs) automated testing is consequently not achievable, since the effort of reimplementing the toolkits functionality is by far exceeding the potential benefit by gained testability.

On the other hand in project specific (non-third party) code (i.e. ScanCAD, as well as employed libraries like SimplewareFramework or Voxelisation) the usage of these code constructs can be minimised. So for instance, by (a) strictly displaying application errors (and warnings) via the application abstraction swApp class (i.e. swApp::DisplayWarning() method), and (b) following non-intrusive user interface guidelines avoiding pop-up windows etc., there are only two pop-up message

box locations in the code. Firstly, in the aforementioned swApp::DisplayWarning() method which can be easily adapted for testing reasons as done by swPseudoApp class; and secondly, for prompting the user if during saving an already existing file with the same name should be overwritten or not.

Subsequently solutions for each type of testing stopper are described first. Because failing wxWidgets assertions, unexpected errors and unhandled exceptions indicate test case failures, abnormal test case terminations are finally contemplated.

1.5.4.1 Modal Dialogs

Modal dialogs run in their own inaccessible "temporary main" message loop waiting for user input. Also, they do not send any idle events. Thus, the on idle time detection based code flow splitting recipe as presented in Section 1.5.2 fails. Typically all standard dialogs (e.g. colour, directory, file selection) provided by wxWidgets are modal, while according to usability guidelines most other modal dialogs need really good justifications, cf. [See01, SL94].

Once again, jfcUnit serves as a reference example. According to its rules and standard examples code using modal dialogs must run in another thread. This thread must not be the current testing thread.

While it is possible, although not recommended at all, to access the GUI in another thread (not being the main thread), some pitfalls remain. By means of paired ::wxMutexGuiEnter() and ::wxMutexGuiLeave() functions the access to the GUI library is correctly synchronised. However, the fact that some wxWidgets code can only be executed by the main thread, for instance ::wxSafeYield() function which is currently already used by progress updates allowing GUI refreshing, makes this not a general solution.

Hints from Williams and Lawrence in [Wil04] and [Law04] respectively to not show the dialog in testing mode by (1) moving out the actual dialog showing code into a separate method, and (2) using a mock object (in testing mode) overriding this method with an empty implementation, are parts of a possible solution. Not actually showing a dialog has no consequences to wxWidgets event handling, as long as the main loop is still run/flushed: processing events for the frame creation, for instance, is nevertheless necessary.

By contrast, Plumlee states that this technique is for testing clients of a dialog, but tests on the dialog itself should be more elaborate and closer to reality [Plu04a]: "Modality couples two concepts - the user can't click the owner window, and the programmer can call the dialog 'like a function', block until it closes, and use its result as a return value [...] That permits the programmer to write less than strictly event-driven code, as a convenience. But it wrecks tests.". For example:

```
void MyDlgClass::Show () {
  if (dialog.ShowModal () == wxID_OK) {
    saveFile (...);
  }
}
```

Developing Plumlee's statements further the solution for modal dialogs is simply showing the dialog *non-modal*. Instead of creating "expensive" mock objects for each modal dialog, a pragmatic globally accessible and configurable swWxGuiTestHelper::Show() method is used. Parameters for this method are the dialog itself, a flag indicating whether to show or hide it, and finally a flag indicating if it is a modal dialog. By default, i.e. in release mode, this method shows the dialog according to the modality flag. However, in testing mode test drivers configure the method to show all dialogs non-modal. Because the FrameFactory (cf. Sections **??** and **??**) creates dialogs, but the modality status is already required at creation time, it is necessary to manage such dialog creations as well. Instead of swFrameFactory::GetInstance()->CreateDefaultDialog() method the corresponding swWxGuiTestHelper::CreateDefaultDialog() method with an option specifying the modality must be used.

Consequently, the code needs refactoring with the goal of becoming event-driven. Otherwise, the control flow in the aforementioned listing directly saves the file, even before the dialog has been "processed" by the user. So, the actual call to show dialogs is independent of testing and release mode:

```
void MyDlgClass::Show () {
  swWxGuiTestHelper::Show (dialog, true, true);
}
```

Completing the refactoring requires driving the code flow by events:

```
void MyDlgClass::EndShow (int retCode) {
  if (retCode == wxID_OK) {
    saveFile (...);
  }
}

...

void MyDlgClassController::OnOkCancelButtonClicked (wxCommandEvent &event) {
  if (event.GetId () == wxID_OK) {
    m_dialog->OnOK (event);
  } else if (event.GetId () == wxID_CANCEL) {
    m_dialog->OnCancel (event);
  } else {
    wxFAIL_MSG ("Illegal event id");
  }
  // Finally tell dialog to end correspondingly:
  m_dlgClass->EndShow (event.GetId ());
}
```

Again, this approach does not work for third party code, that is, for testing the latter either avoid executions of such code bits, or apply the so-called "timed dialog ender" technique which is presented with respect to message boxes in the next section. Certainly, this technique has strong limitations in this context by allowing solely the controlled ending of dialogs without any other dialog interactions.

1.5.4.2 Message Boxes

In wxWidgets message boxes (wxMessageBox class) are internally made up of wxMessageDialog classes on all platforms. Thus, they are in fact modal dialogs and, in principle, everything stated in the previous section can be applied on them. By comparison, processing message boxes is much easier because only a limited set of interactions is possible: message boxes consist of one or more buttons (e.g. OK, Cancel, Yes and No), and each button press is equated with closing the dialog with the corresponding return value. Therefore testing of message boxes is easier. As a side note, jfcUnit asks for the same testing technique than for modal dialogs in general.

Internally wxWidgets does not use message boxes very often in code which will find its way into the release build. One of the few locations is the error handling in the Document-View architecture to query, if an existing file should be overwritten for instance. Likewise, ScanCAD application and SimplewareFramework library only use message boxes in two cases as aforementioned: showing application warnings is the important one in the following. By contrast, defensive programming functionalities which are only there in debug builds make heavy use of message boxes, since they catch the developer's eye easily in case of warnings or errors.

A solution for message box blockings in [Ghe03] is treating message boxes as the simplest form of modal dialogs, and suggests testing them with mock message boxes.

The approach based on ideas in [Won04] is completely different in that it detects and automatically responds to user input requests from running applications. Specific captions allow identifying dialogs, and therefore wxWidgets message boxes. Using a timer avoids blocking the CPU by looking only after specified intervals for dialogs, instead of looking continuously. Subsequently, found dialogs can be forced to close with certain return or ending codes, hence the technique is called "*timed dialog ender*" in the following. Owing to its non-intrusive manner it is also functional for third party code using modal dialogs. Unfortunately, the current implementation works only under Microsoft Windows, maybe under Unix-like OSs, but very likely not on Mac due to the Apple Human Interface Guidelines [App05].

The timer (which can only be used from the main thread currently) is carrying on to fire events even when the application (and therefore the main thread) is actually blocked. Interestingly, trying to destroy the dialog with brute force via DestroyWindow() function (Microsoft Windows Platform SDK) is detected in debug mode by wxWidgets' excellent defensive programming style. While the dialog vanishes, the testing application also stops processing any user input afterwards. By contrast, EndDialog() function (Microsoft Windows Platform SDK) "softly" closes modal dialogs and wxWidgets message boxes with a specifiable return code.

While this technique can solve message box blocking, it has at least two drawbacks: (1) the caption of the dialog must be well-known in advance, and (2) it lacks platform independence.

On a second thought the quite limited usage of message boxes leads to a neat solution: Pop-up

message boxes are solely (ignoring the single file overwrite prompt) caused by noticed application errors and warnings leading to calls of swApp::DisplayWarning() method. Of course, sometimes testing should test the occurrence of exceptional cases, thus, requiring that a message box will pop up. Again, such provoked or expected warnings can be dealt with by the timed dialog ender. In consideration of the fact that both, (a) occurrence of unprovoked warnings, and (b) missing to occur provoked warnings, indicate failing test cases, this technique is doomed to fail. There is no way to create a general timed dialog ender, so each potential warning would need to be initially taken into account with a separate specialisation based on the caption.

So, the solution is to directly interfere in swApp::DisplayWarning() method in testing mode (see Listing 1.12). This also provides more information to be added to the test case failure message about what went wrong. Nevertheless there is an identified need of discriminating between provoked (initiated by test cases) and unexpected warnings (= real test case failures). A registry managing provoked warnings whose number is by far exceeded by the vast number of potential warnings provides this discrimination.

```
bool sccScanCADApp::DisplayWarning (const wxString &caption, const wxString &
        message, bool canBypass) {
  //[WxAppGuiTesting] - Allow detection of unexpected/unprovoked warnings:
  #ifdef SW_USE_WX_APP_GUI_TESTING
    // Don't care about provoked or not, just return false:
    swWxGuiTestHelper::IsProvokedWarning (caption, message);
    return false;
  #endif

  ...
```

Listing 1.12: Overridden application error handling for test case failure detection.

Each time a warning is detected an iteration over the registry returns the discrimination information (see Listing 1.13). Allowing test cases to check whether or not a provoked warning has occurred requires labelling it (cf. SetWarningAsDetected() method). Once more, for the purpose of avoiding side effects clearing the registry for each test case is important. Finally, a time out interval for provoked warnings reduces the risk of mixing up identical warnings caused at very different points in time (or test case executions).

The eventual usage of (A) the timed dialog ender, as well as (B) the preferred method of interfering early and utilising the provoked warning registry is shown in Listing 1.14, where a warning is expected during test case execution.

1.5.4.3 Failing Assertions in Debug Mode

In debug mode assertions allow following defensive programming style. For the sake of efficiency they are not part of release versions. Failing assertions interrupt executing test cases via message

```
bool swWxGuiTestHelper::IsProvokedWarning (const wxString &caption, const
        wxString &message)
{
  bool isProvoked = false;

  swWxGuiTestProvokedWarningRegistry &provWarningRegistry =
          swWxGuiTestProvokedWarningRegistry::GetInstance ();
  const swWxGuiTestProvokedWarning *warning = provWarningRegistry.
          FindRegisteredWarning (caption, message);
  if ((warning != NULL) && (provWarningRegistry.IsRegisteredAndInTime (*
          warning))) {
      // It is a provoked warning -> label it as detected:
      provWarningRegistry.SetWarningAsDetected (*warning);
      isProvoked = true;
  } else {
      // Unexpected warning -> test case failure:
      wxString failMsg = wxString::Format ("Caption \"%s\", message \"%s\"
              occured", caption, message);
      swWxGuiTestHelper::AddTestFailure ("", -1, "Unexpected swApp application
              warning detected", failMsg);
      wxTheApp->ExitMainLoop ();
  }
  return isProvoked;
}
```

Listing 1.13: Checking for provoked warnings.

boxes (with the constant caption "wxWindows Debug Alert"). They ask whether the user wants to abort the program, continue or continue ignoring any subsequent assert failures. Thus, during implementation they facilitate localising errors by offering to jump to the source code location of the failing assertion. Subsequently the whole debug environment is set up providing a full stack trace for example.

Due to the close relation with message boxes, and the well-known constant caption, one global timed dialog ender is again a first possible solution. However, interfering earlier is again preferable (cf. the final message box blocker solution), i.e. before the message box is created. By overriding the default assertion handling (wxApp::OnAssert() method; see Listing 1.15) in swWxGuiTestApp class, the whole assertion specific context information is accessible. Thus, the test case failure message is very meaningful by including assertion condition, an optional assertion failure message and exact assertion location (file name and line number). For instance, a test run may end up with:

```
.....F...

D:\CVS\ScanCAD\Cxx\View\sccmultipleview.cpp(752):Assertion
Test name: sccTst::sccDatasetBrowserGuiTest::testChangeColour
wxWidgets assert failure
```

```
// (A) Old with timed dialog ender:
/*
swWxGuiTestTimedDialogEnder msgBoxKiller (2000, _("Direct CAD Voxelisation
        Warning"), IDOK);
*/
// (B) New with provoked warning registry:
swWxGuiTestProvokedWarning warning (_("Direct␣CAD␣Voxelisation␣Warning"),
        NULL, 3);
swWxGuiTestProvokedWarningRegistry &provWarningRegistry =
        swWxGuiTestProvokedWarningRegistry::GetInstance ();
provWarningRegistry.RegisterWarning (warning);

... // Execute code which should lead to a warning

// Check if warning message box has really popped up:
// (A) Old with timed dialog ender:
/*
CPPUNIT_ASSERT_MESSAGE ("No direct CAD voxelisation warning detected",
        msgBoxKiller.GetSuccess ());
*/
// (B) New with provoked warning registry:
CPPUNIT_ASSERT_MESSAGE ("No␣direct␣CAD␣voxelisation␣warning␣detected",
        provWarningRegistry.WasDetected (warning));
```

Listing 1.14: Detection of provoked warnings in test cases.

```
- Assert "wxAssertFailure" failed: TODO: support this VTK dataset type as well

Failures !!!
Run: 8  Failure total: 1 Failures: 1 Errors: 0
```

1.5.4.4 Pop-up Menus

Similar to message boxes, pop-up menus only return control after dismission by the user. Apparently at least in Java based GUI testing they ask for special treatment, because handles to them cannot be retrieved in the same way as for frames or dialogs. Thus, jfcUnit's so-called Window-Monitor, which detects all window creations in the AWT event queue already, must be started before the first GUI interaction.

In analogy to the modal dialog blocker solution, but in consideration of the lack of an intermediate technique like showing non-modal, the only way is to not show pop-up menus at all in testing mode. The configurable swWxGuiTestHelper::PopupMenu() method provides this functionality. For the purpose of testing the pop-up menu itself, e.g. enabled or checked status of menu items, access to pop-up menus is vital. In contrast to dialogs this is not supported due to the fact that wxMenu class is not subclassing wxWindow class. Instead of providing public getter methods for each pop-up menu violating the information hiding principle, caching them in a string key based hash map in the course of calling swWxGuiTestHelper::PopupMenu() method is a clean solution. So, showing all

```
void swWxGuiTestApp::OnAssert (const wxChar *file, int line, const wxChar *
        cond, const wxChar *msg) {
  if (swWxGuiTestHelper::GetPopupWarningForFailingAssert ()) {
#ifdef __WXDEBUG__
      wxApp::OnAssert (file, line, cond, msg);
#endif // __WXDEBUG__
  } else {
    wxString failMsg = wxString::Format ("Assert \"%s\" failed", cond);
    if (msg != NULL) {
      failMsg << _T(": ") << msg;
    }

    swWxGuiTestHelper::AddTestFailure (file, line, "wxWidgets assert failure"
            , failMsg);
    wxTheApp->ExitMainLoop ();
  }
}
```

Listing 1.15: Overridden default assertion handling for test case failure detection.

pop-up menus via this method is the only necessary change supporting normal release and testing mode. In turn test cases can use swWxGuiTestHelper::FindPopupMenu() method to query specific pop-up menus.

1.5.4.5 Unhandled Exceptions

As already pointed out, unhandled exceptions should never occur in an application due to defensive programming. Try-catch blocks around each potentially exception throwing piece of code catch all exceptions. If exceptions are not caught, the application will crash (in release mode). For the purpose of carrying out any clean up tasks wxWidgets allows developers to override wxApp::OnUnhandledException() method. Unhandled exceptions occurring in the course of running the application will result in calls of this method.

However, this does not happen in testing mode: swWxGuiTesting, or in fact the embedded CppUnit testing framework, catches all exceptions a priori for test case success checking, i.e. uncaught exceptions already lead to test case failures by default.

1.5.4.6 Abnormal Test Case Terminations

Failing wxWidgets assertion, unexpected warning and unhandled exceptions are all indicating test case failures. Due to the assumption that the first detected test case failure will lead to aftereffects, terminating the test case with a meaningful error message as soon as possible after the failure detection is crucial. Waiting for user input to potentially recover from the error is no option in automated testing. Thus, testing must resume after test case failure detection by running the remaining test cases.

Unfortunately, in contrast to test case termination based on failing checkpoints in test routines, problems emerged during developing termination mechanisms for the aforementioned test case failure indicators. Test routine checkpoints are basically CppUnit specific assertion macros which are throwing exceptions in case of failure. These exceptions contain detailed error messages and are caught by the testing environment, or more precisely by test case executors, in turn.

Influenced by these macros, the nice idea of throwing a special exception only being know and therefore catchable by the test case executor (= the so-called protector) to signal test case failures was born. Of course, this also necessitates an additional implementation rule: either never catch all exceptions independent of their type via the `catch (...)` construct, or forward it again, if it is the special exception exptected by the test case protector. However, after implementing the corresponding exception, protector, test result and test runner classes in some cases unacceptable irregularities occurred: while these abnormal test case terminations are always happening during event handling (initiated by flushing the event queue from test cases), it seems that the event handling is not finished properly in all cases and the application sometimes can therefore not be closed leading to an infinite loop. Consequently, test runs will neither finish, nor produce any test result output.

It is assumed that introducing exception handling in the event handling of wxWidgets can solve this issue. Without changing wxWidgets it is necessary to finish flushing the event queue, i.e. to wait until the currently pending events are processed (cf. Listing 1.16). Thus, test cases are not at all stopped immediately, potentially leading to further failures. Of course, this should never happen in a released application, but maybe subsequent failures are legitimate during testing applications... See Figure 1.9 for a class diagram of all involved classes with respect to abnormal test case termination.

```
int swWxGuiTestHelper::FlushEventQueue () {
  // Reset test case failure occurrence variables:
  s_fileOfFirstTestFailure.Clear ();
  s_lineNmbOfFirstTestFailure = -1;
  s_accTestFailures.Clear ();

  ... // Run main loop temporarily

  // If a failure has occured, throw exception:
  if (!s_accTestFailures.IsEmpty ()) {
    wxASSERT (s_warningAsserter != NULL);
    s_warningAsserter->FailAssert (s_fileOfFirstTestFailure,
            s_lineNmbOfFirstTestFailure, s_shortDescriptionOfFirstTestFailure
            , s_accTestFailures);
  }
}
```

Listing 1.16: Overridden default assertion handling for test case failure detection.

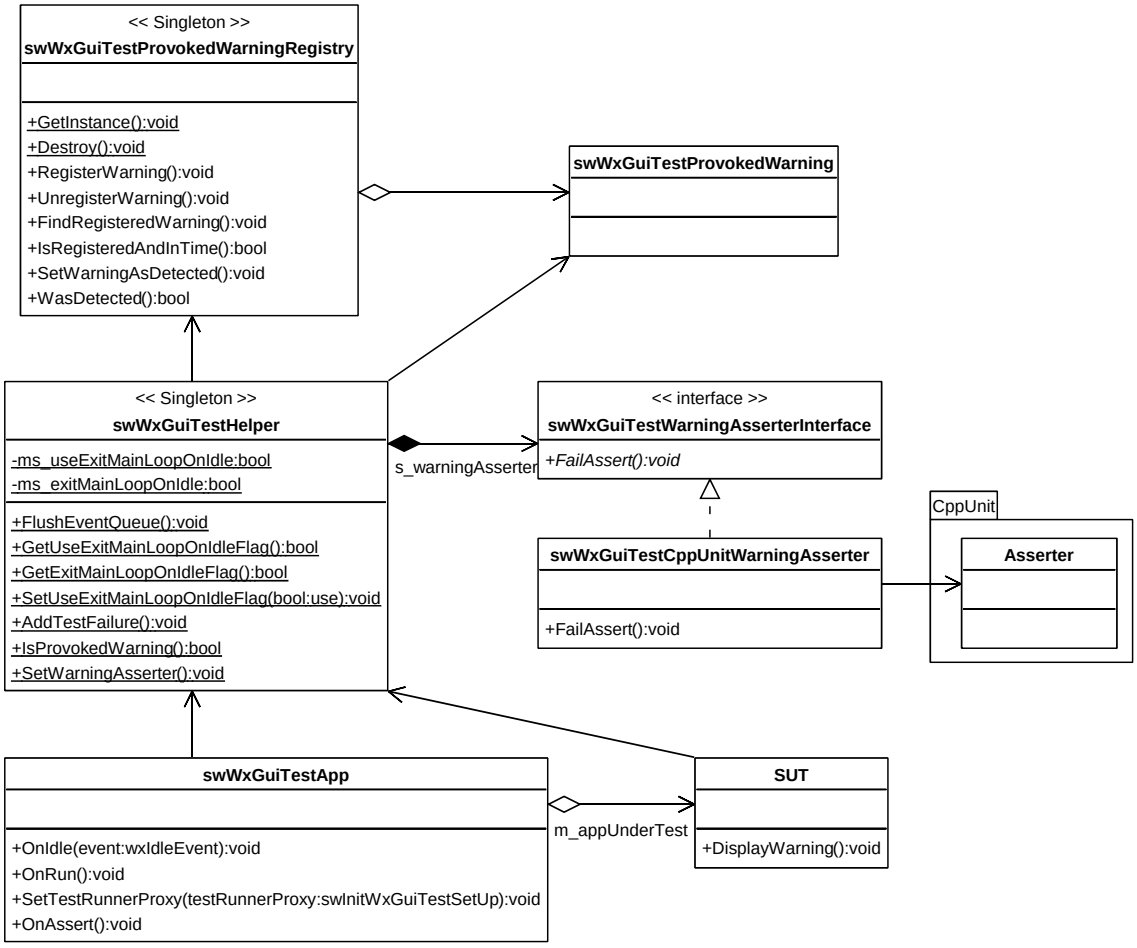


Figure 1.9: Class diagram: Abnormal test case termination.

1.5.5 Implementing GUI Test Cases

As pointed out in the previous section using the developed swWxGuiTesting framework for actual GUI and application testing poses some problems or potential testing blockers. Summarising the required changes for future SUTs will cover this issue. Subsequently, helper functionality facilitating the concrete test case implementation is presented. This comprises firstly techniques to make tests during the test implementation temporarily interactive for visual checks, and secondly a helper class for the unusual and partly inconsistent simulation of events.

By and large, the implementation of GUI and application test cases is relatively easy and familiar owing to a number of facts: (a) wxWidgets is internally not modified at all, (b) test drivers as well as test case skeletons are automatically generated based on templates with wxCRP (cf. Section **??**), and (c) appropriate abstractions for every non-standard task make the life of developers quite comfortable. To demonstrate this a simple application testing example is shown in Section A.1.6 in the appendix: importing an STL file via the corresponding dialog.

1.5.5.1 Required Changes of Future SUTs

A perfect testing tool does not require any changes of the application in order to turn it into an SUT. Unfortunately, swWxGuiTesting is a rather pragmatic solution, but with the clear advantage of being familiarly embedded into "normal" development: (a) no extra tool or scripting language is required; (b) if desired, failing test cases or assertions stop the tests and open the debugger at the failure location; and (c) implementing test cases is easy and familiar as stated above. Thus, due to the framework's pragmatism developers must carry out a few modifications, and keep mechanisms linked with the employed idle time detection approach in mind. These items fall in two categories: changes in relation with the application "main" class or instance vs. changes in all classes. All modifications are configurable since the code does not have to be changed to switch between testing and release mode, but the (automatically generated) test driver contains the configuration needed for testing.

To recapitulate, the application main file must support the application instantiation macro redefinition as explained in Section 1.5.3. Moreover, overriding MyApp::DisplayWarning() method according to Section 1.5.4 and Listing 1.12 provides the necessary distinction between provoked and real errors in testing mode, whereas the latter must result in test case failures.

On the other hand modifications in all files are necessary with respect to:

- *Modal dialogs:* Showing (modal) dialogs should be initiated by means of swWxGuiTestHelper:: Show() method, instead of the standard way via dialog/frame−>Show/ShowModal() methods. Boolean parameters specifying show vs. hide and modal vs. non-modal allow showing modal dialogs non-modal in testing mode. Because the FrameFactory (cf. Section **??** and **??**) creates dialogs, but already at dialog creation time the modality status is required, managing such dialog creations by means of swWxGuiTestHelper:: CreateDefaultDialog() method is crucial. Finally, only event-driven code is testable. Of course, all these changes are only necessary for code bits containing the display of modal dialogs intended to be tested; see Section 1.5.4 for details.

- *Pop-up menus:* Similar to modal dialogs pop-up menus are blocking, so they must not be shown in testing mode. Supporting this requirement is asking for calling swWxGuiTestHelper::PopupMenu() method instead of wxWindow::PopupMenu() method.

- *Message boxes:* For the purpose of communicating warnings and errors the application abstraction's swApp::DisplayWarning() method shall be used. In probably all other cases, message boxes should be avoided anyway due to usability guidelines.

Finally, owing to the idle time detection centric approach care has to be taken in case of non-standard code dependent on idle events.

1.5.5.2 Temporary Interactive Tests

In [Plu04b] Plumle reveals the importance of visually inspecting GUIs and checking GUI layout and event handling during test case implementation quite plainly. He calls this "temporary interactive tests". In contrast to his approach which requires commenting in and out this feature, a property – configurable by the test driver – allows disabling interactivity for regression test runs completely, as well as enabling it during test case implementation (cf. swWxGuiTestHelper::SetDisableTestInteractivity() method).

The easiest technique is basically using one of the automated testing blockers in the test case routine, namely popping up a message box prompting the user if she wants to continue automatic GUI testing. While this is trivial and supported via the swWxGuiTestHelper::BreakTestToShowCurrentGui() method consisting of no more than three statements (including testing for the aforementioned interactivity enabled flag), it is not sufficient in general: the modal dialog underlying the message box prevents any interaction with and manipulation of the GUI. Thus, not even re-layouting of dialogs caused by dialog size changes can be checked.

If visual inspection without any interaction is not enough, real (temporary) interactivity is demanded. As already indicated the latter supports running the GUI (almost) as in release mode and checking in particular the correct event handling setup. Of course, in theory, this interactivity poses the threat of side effects: the user (= the person who is developing the GUI test, so she is trustworthy) is interacting with the real GUI under test and could manipulate it disallowing a proper continuation by closing the main frame for instance.

The implementation for real interactive tests consists of:

1. Inverting the flag which will cause an exit of the main loop on idle time detection during testing ($\Rightarrow$ do not exit main loop on idle).
2. Creating and showing an extra non-modal dialog with prompting the user for continuation of testing; this allows interaction with the GUI under test, as well as the extra dialog to stop the temporary interactivity and resume testing.
3. Starting the main loop.
4. Handling the continuation event for the extra dialog in order to resume testing by (a) hiding the dialog, (b) exiting the main loop based on event queue flushing, and (c) restoring original flag(s).

As simple as the idea of temporarily interactive tests may seem, as helpful it has proven to be in the course of implementing GUI unit tests. Only this comparatively small feature constitutes a big step forward to TDD of GUIs and applications, i.e. the "methodology" underlying its idea: test first user interfaces.

1.5.5.3 Event Simulation Helper

The simulation of events is probably the only unusual coding within GUI test cases, even though it is perfectly legal standard wxWidgets code. Normally developers only come in contact with such low-level event code, if they are developing their own non-standard widgets. In all other cases (probably 99% of all written code using wxWidgets) this is hidden by the toolkit. Therefore an event simulation helper class represents a useful abstraction in GUI test cases to developers.

Another motivation for this abstraction is the partly inconsistent behaviour of different controls due to simulated events. I.e. different controls require different procedures to simulate events for the same type of control state modification. Thus, the helper class (a) frees the developer from the need to know the individual variations, and (b) hides the actual implementation in the optimistic case of corrections in future wxWidgets versions with respect to event simulating inconsistencies.

Some examples of inconsistency are:

- Tree control events (tree item selection as well as right clicks) must be processed immediately, instead of being posted; otherwise the event's properties are getting invalid paradoxically.
- Events to change selected pages of notebook widgets are not processed correctly, instead accessing the actual notebook component to change the selection via a setter method is necessary.
- Changing values of text controls creates events in opposition to changing values of spin controls which does not.

As already mentioned in Section 1.5.1.3, only high-level events can be simulated. This is mainly a result of wxWidgets' platform independence requirement: In release mode manipulating a control by users leads to platform specific (low-level) events. wxWidgets in turn handles these platform specific events and converts them into platform independent control specific events. Subsequently, the platform independent standard wxWidgets event handling is resuming.

Hence, in testing mode most controls have an inherent distinction between GUI state changing and actual event generation (and therefore event handling). In contrast to release mode where the changing GUI state fires the event, i.e. the control has already the correct GUI state, it is often necessary to simulate both distinctive features: If the code flow depends on the generation and handling of events, but also depends on the control's correct GUI state, (1) the control's value has to be changed (usually by means of SetValue() method), and (2) the corresponding event has to be created and fired. Another reason for this "double" simulation is the test case implementation developer's desire to check the control's correct GUI state when tests are temporarily interactive.

While the Java GUI testing framework jfcUnit offers a mature and OO class compound built around the TestHelper class (e.g. enterClickAndLeave() method), the chosen pragmatic approach in swWxGuiTesting initially only provides helper methods for each demanded GUI control interaction

in a single class. For instance, it supports clicking on buttons, or selecting an item from a listbox. As a matter of course each event simulation method from the helper class swWxGuiTestEventSimulationHelper is tested automatically by GUI test cases.

Finally, it should be noted that according to the wxWidgets documentation a simple two line event processing loop is allegedly supported. Consequently, the code flow control splitting implementation is potentially much easier and clearer. Unfortunately this event loop is not working correctly, i.e. it is unable to process the simulated events.

1.5.6 Capture & Replay

In this section C&R functionality, taking the requirements from Section 1.2 – such as preventing the need to learn a new and proprietary scripting language – into account, is added to the swWxGuiTesting testing framework. Most commercial GUI testing tools employ C&R techniques to aim at increased the test case development productivity. As pointed out in Section 1.1.6 applying C&R consists of three steps leading to (mostly proprietary) high-level programming language test scripts: (1) record (capture) user interaction in test script, (2) add checkpoints and assertions to test script in between recorded user interactions, and (3) replay test script (repetitive replaying achieves regression testing, cf. Section 1.1.4).

In contrast to step #2, which is context and test goal dependent, and must be therefore carried out by the test case developer after the capturing phase, step #1 and #3, capturing and replaying respectively, can be fully automated.

This section resumes with the aims & requirements of developing C&R technology and the chosen usage workflow, before concluding with vital aspects in detail, as well as the overall solution for C&R of normal GUI and VTK interactions.

1.5.6.1 Aims & Requirements

In general C&R development raises the following questions:

1. What to capture?
2. How to capture?
3. How to save the captured data?
4. How to add checkpoints?
5. How to replay the saved captured data augmented with checkpoints?

The answer to the first question is easy and unsubtle:

> Any information required to replay captured events by means of simulation, i.e. event details plus data to re-identify controls required for event simulation, must be captured.

For the sake of maintainability and user-friendliness this aim is complemented by a further vital goal:

> Moderate changes of the GUI without necessitating the re-capturing of existing test cases must be considered, supported and allowed.

Deriving concrete requirements from these items leads to (a) capture phase objectives:

1. Starting the GUI/application under test
2. Controlling the capturing process by means of a capture dialog providing interaction controls
3. User interaction detection and handling (= event handling/filtering; depends on next item)
4. Identifying affected controls

, and (b) replay phase objectives:

1. Starting the GUI/application under test
2. Re-identifying affected controls
3. Event simulation
4. Evaluate checkpoints and assertions

Hereby, the capture phase objectives give general answers to question #2 ("How to capture?"), while replay phase objectives tackle the last question ("How to replay...?").

Questions #3 ("How to save the captured data?"), #4 ("How to add checkpoints?") and again the last question are all strongly linked with the test script format. In theory, each C&R phase needs its own persistence scheme:

1. Persistence of capturing output and captured data
2. Intermediate (?) format for adding checkpoints
3. Final (?) format for replaying captured data augmented with checkpoints

Standard `C++` GUI/application unit test cases employing wxWidgets being executed from standard CppUnit test drivers are making up the current test case format (cf. Section A.1.6 in the appendix). However, utilising an XML test case format provides the advantage of allowing further automatic (post-)processing by means of user-friendly GUI-based tools. Thus, even people

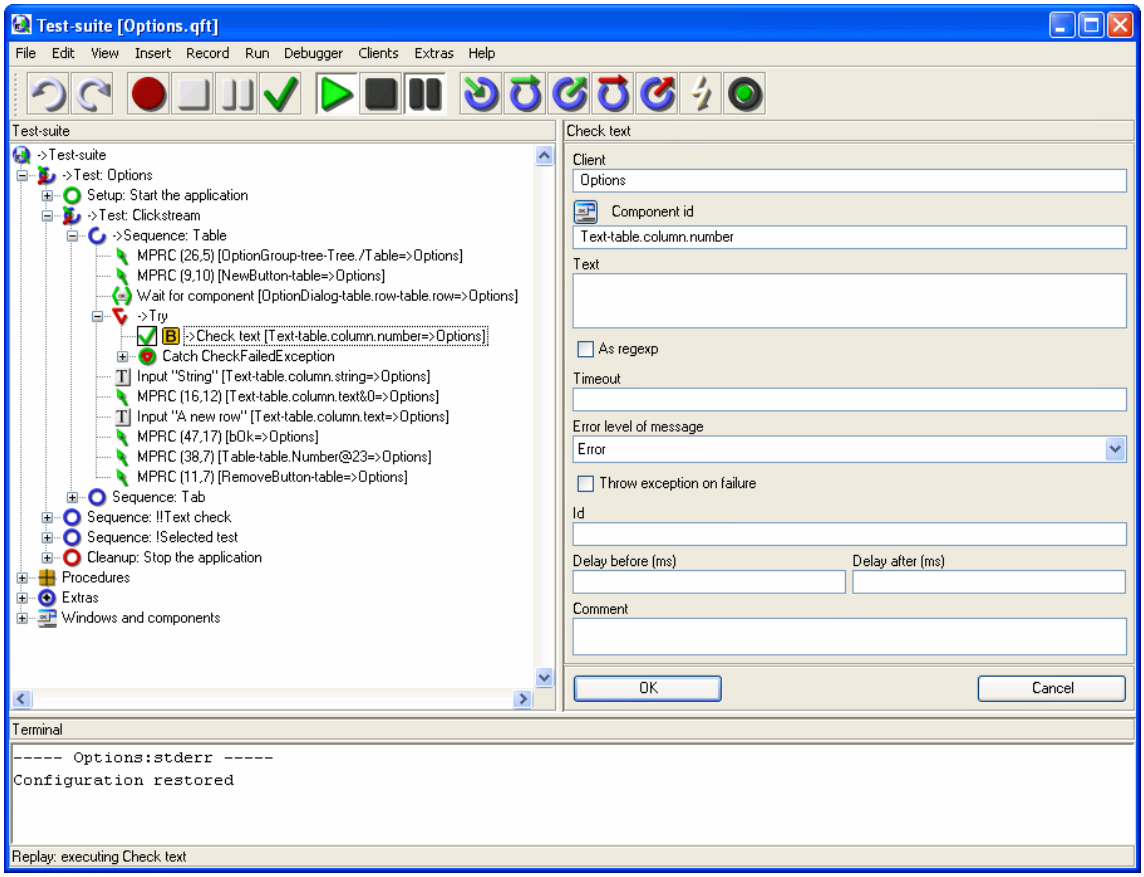


Figure 1.10: Java GUI test tool qftestJUI.

from specialist departments – who know their particular requirements the best – without specific software development background may be able to create complete test cases without the help of software developers. This saves communication overhead and minimises the risk of misunderstandings in communicating their demands to the otherwise badly needed software developers. While the commercial Java GUI test tool qftestJUI [Qua05] (see Figure 1.10) is an example for such a tool, it is still too software development oriented.

Moreover, XSLT technology can automatically convert these XML test case files into the current GUI unit test case format, i.e. standard `C++` based GUI/application test cases to be executed by normal test drivers.

A more direct and at least short term option is saving the captured user interaction directly/only in the current test case format. Nevertheless, using XML format persistence for test cases, plus allowing the automatic generation of the current test case format, is preferable in the long run. Obviously, the generated `C++` code should follow certain guidelines (e.g. line breaks) to allow for easy readability and maintenance, besides being compilable with only little modifications.

Finally, allowing moderate changes of the GUI without having to re-capture the test case mainly poses requirements on re-identifying the affected controls symbolically.

1.5.6.2 Usage Workflow

Obviously, the capture phase is the most complex part of C&R. In principle it comprises all objectives of the replay phase, but it must also prepare the latter by processing the available information during its activity. (Event simulation is mandatory for current ordinary GUI/application test cases too, and its solution is therefore outlined in the previous sections.)

As this section will reveal, the developed solution for the first identified objective, namely starting the GUI or application under test, has significant effect on C&R's usage workflow.

Basic approaches to start the SUT are:

- Yet another build target: in order to allow mere GUI testing besides application testing yet another pseudo capture application class, similar to the swWxGuiTestApp for testing is necessary.
- Make the capturing a special "pseudo" test case and provide a text control for the filename (or test case name) in the capture dialog: non-developers can capture at least acceptance test case skeletons.
- Going one step further to a special "pseudo" test case is starting from within a GUI/application test case: bootstrap the GUI test case by replacing the capturing initiation call (in the test case) with the concrete subsequent user interactions (= events to simulate).

The last approach, *bootstrapping*, was chosen due to its elegance because it also solves questions #3 ("How to save the captured data?"), #4 ("How to add checkpoints?") and #5 ("How to replay the saved captured data augmented with checkpoints?") with only moderate development effort, whilst providing high usage comfort. The fundamental bootstrap workflow is presented on the basis of the following pseudo test case modfications:

1. Initial test case – before capture phase:

```
test () {
  showGUI (...);
  CAPTURE
}
```

2. After capture phase → bootstrapped test case:

```
test () {
  showGUI (...);
  findSomeControl (...);
  simulateSomeEvent (...);
  findSomeControl (...);
  simulateSomeEvent (...);
  ...
}
```

3. After manually adding checkpoints → complete test case:

```
test () {
  showGUI (...);
  someCheckpoint (...);
  findSomeControl (...);
  simulateSomeEvent (...);
  someCheckpoint (...);
  findSomeControl (...);
  simulateSomeEvent (...);
  someCheckpoint (...);
  ...
}
```

Thus, a new test case adheres at any time and in any C&R phase the current test case format. Therefore it seamlessly integrates into the current test execution, i.e. the *replay* phase is transparent by being identical to executing normal GUI test cases simulating the captured events one after the other as emitted to the test case during bootstrapping. Moreover, test case developers can apply exactly the same technique, tools and knowledge as before; but they benefit from the hugely increased productivity in creating test cases.

One of the bootstrap approach specific core features is the location of the capturing initiation call in the test case. By making the initiation call a macro named CAPTURE, which employs the C/C++ standard macros __FILE__ and __LINE__ the corresponding test case source location can be easily figured out. As aforementioned, the concrete subsequent user interactions making up the events to simulate will replace this line in the end.

Although the bootstrap process could directly manipulate the test case source file, two reasons led to the decision to create the bootstrapped test cases in the shape of backup files:

1. Sometimes backups of original files should be kept out of whatever motivation, e.g. allowing to undo bootstrap capturing operations.

2. During implementation of GUI test cases it is quite likely that developers are debugging the test case file which is currently bootstrapped; due to modifying the source code breakpoint locations are getting wrong, and debugging as such may therefore become confusing.

Besides starting the SUT, controlling the capturing process by means of a *capture dialog* is the second big aspect with respect to the C&R usage workflow. The capture dialog (see Figure 1.11) provides interaction controls to start, pause and quit the user interaction recording. Furthermore, it allows adding comments in between captured user interactions to the test script for structuring the test case, or leaving checkpoint adding instructions for instance.

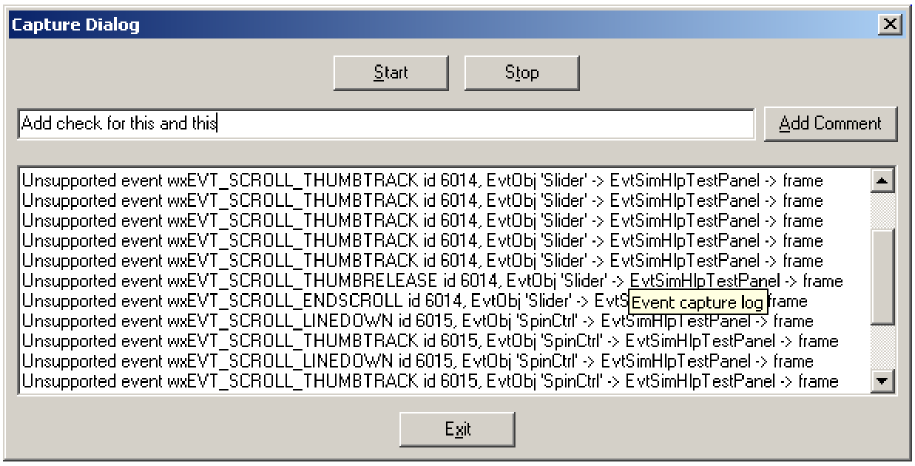


Figure 1.11: Capture dialog.

1.5.6.3 User Interaction Detection

As already pointed out, capturing serves the purpose of recording user interaction. This can be split into user interaction detection and the corresponding processing to allow a later replay; these two parts are presented in this and the following section.

Once again, the closeness of the swWxGuiTesting framework to the SUT with respect to design, implementation and runtime aspects allows a graceful solution. Any user interaction with the GUI under test results in the generation of events. Thus, detecting user interaction for C&R can be seen as just another type of event handling: a global event handling in comparison with ordinary event handling of GUIs which is mostly locally restricted (e.g. one event handler class is responsible for one dialog).

wxWidgets' documentation recommends pushing event handlers on top of the GUI for the latter (locally restricted event handling). Similarly to Section 1.5.1.4 this concept is discarded, because of the highly likelihood that another piece of code is doing the same again (i.e. pushing another event handler on top of this one). So, modifying wxWidgets in order to support event handler stack change notifications and stack re-orderings would be necessary.

The second and eventually implemented approach is based on wxTheApp::HandleEvent() method, which is called by each event handling according to the documentation. However, this method does neither exist in the employed version of wxWidgets, nor is it part of any newer version. Thanks to wxWidgets being open-source it was easy to find out that in fact wxApp::FilterEvent() method provides this functionality of processing all events instead. (Only very recently this mistake has been corrected in the documentation.)

Overriding wxApp::FilterEvent() method in swWxGuiTestApp class (the GUI testing framework's application instance stub) allows detecting any user interaction and thus, capturing events. Actually, this responsibility is delegated to the swCREventCaptureManager class as depicted in Figure 1.12.

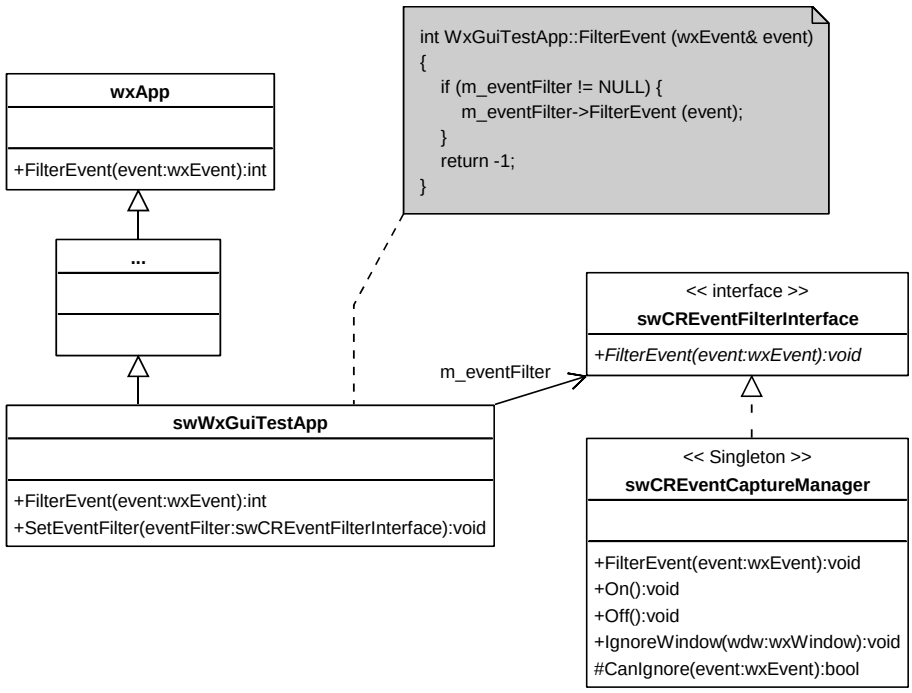


Figure 1.12: Class diagram: User interface detection.

The name of the method already indicates that its real purpose is enabling applications to ignore or preempt some events. Unfortunately, but not really surprisingly, this event filtering is executed numerous times caused by:

- Calling the method several times with the same event as argument leading to duplicate events, because parent event handlers are automatically investigated as well in most cases.
- Interaction with the capture dialog which obviously also generates events.
- Generation of some system events for any window and GUI control creation.
- Generation of much more system events due to necessary refresh and repaint directives.

Identifying duplicate events appears trivial after studying wxEvent class, which is the parent class of all events and consequently used as input parameter to the event filtering method. However, its focused timestamp property does not help in recognising unique events, since it is rarely set correctly. So, a potentially risky event address comparison is utilised to carry out this task.

Two little helper methods in swCREventCaptureManager class fulfil the job of ignoring events stemming from capture dialog interactions based upon window hierarchy inspection of the event initiating and/or affected controls.

Finally, filtering out system events stimulated by window and GUI control creations, or repaint directives for instance, requires gathering information about the concrete event types. One way is

using the C++ Runtime Type Information (RTTI), but besides being slow in general, the wxEvent class reference typed argument has another serious disadvantage: in opposition to pointer variables for which cast operators return NULL, references throw bad_cast exceptions in case of trying to cast to wrong types. Hence, the code for filtering out the big number of different system events is getting extremely long and unreadable.

Instead employing wxEvent::GetEventType() method provides fast access (as aforementioned numerous events have to be filtered) to the concrete event type in the shape of an integer. Unfortunately, due to recent wxWidgets modifications the event types are not constant anymore, but instantiated in the order of runtime usage. Thus, the source code is less compact because it must use if in place of switch statements, and there is no way to exclude all system events as a continuous event type range.

As already pointed out in the previous section the capture dialog allows starting, pausing and quitting the user interaction recording. While these events are naturally ignored by the capturing as outlined above, they also make use of functionality provided by swCREventCaptureManager class to switch the event filtering (and therefore the user interaction recording) on and off.

1.5.6.4 Identifying Affected Controls

After detecting user interactions in the shape of events, processing these events for the purpose of allowing a later replay is the complementing second vital task. Due to the chosen C&R approach – preparing the replay phase fully – the capture phase takes the bulk of responsibility: i.e. the generated test cases with the focus on simulating events can be executed directly.

Event simulation necessitates knowledge about which GUI control initiates or is affected by such events. Thus, capturing must gather and provide this knowledge. Captured events typically contain properties with the numeric ID of the control, plus an object (usually a control) that the event was generated from, or should be sent to. Numeric IDs must not be utilised in identifying controls for replay, since (a) wxWidgets automatically assigns numeric IDs to controls without any foreseeable rule, and (b) any wxWidgets GUI modification will lead to a different numeric ID assignment for future playbacks (= test case runs) with the utmost probability. The latter reflects the "soft" C&R objective allowing moderate changes of the GUI without having to re-capture the test script which mainly poses requirements on symbolically identifying the affected controls again.

Although controls can be identified by all its properties, this is deemed to laborious besides problematic issues concerning the aforementioned moderate GUI changes; and the opposite case of solely taking the control's pointer variable address is more than naive. Fortunately, wxWidgets provides a means of naming each control which – with some development discipline – permits identifying the event initiating and affected controls by *symbolic* names.

In order to avoid any side effects or conflicts between equally named or similar controls, the controls must be therefore *uniquely* identified.

GUIs are hierarchical compositions using recursion. For example, with the exception of wxMenu* instances all wxWidgets GUI controls are wxWindow instances. Moreover, certain GUI controls (namely wxDialog, wxFrame, wxPanel and wxNotebook classes), so-called window or GUI control containers, can hold recursive compositions of other GUI controls.

Softening the symbolic uniqueness condition to uniqueness with respect to GUI control containers in combination with saving the whole GUI control hierarchy allows the aforementioned moderate GUI changes. So, if something changes, the look-up hierarchy level can be automatically changed. Nevertheless, this multilevel approach complicates the control identification:

1. Inspect window hierarchy to key out appropriate container
2. Identify this container by unique symbolic name
3. Identify event initiating and affected control within this container by symbolic name, being unique within this container

One of the issues is now how to identify the correct container level on which to strive for unique names. This requires some knowledge about the hierarchy of used containers created (a) programmatically based on the frame factory (e.g. main frame; cf. Section **??**), and (b) descriptively via XRC. Typically only the main frame and SimplewareFramework specific non-standard widgets are created programmatically; so menu item selection from the main frame's menu bar is tangled for instance.

On the other hand, as already pointed out, all remaining GUI elements shall be designed using XRC. Furthermore, for the sake of reusability and concurrency (wxDesigner's project file format is binary and no full reverse engineering of XRC files is supported yet) each container should be put in it's own XRC file.

Consequently, uniqueness on the XRC root node level seems appropriate in general, resulting in picking XRC root node containers by default. This of course requires recognising XRC nodes based on the directly available numeric ID of controls. Although wxWidgets offers a lot of helper functionality, it is not generally sufficient in this inverted problem: a control is an XRC control, if applying the XRCID macro on its symbolic name returns its numeric ID (assuming the uniqueness condition is globally fulfilled, and anticipating the condition that by default the XRC system creates controls by setting their name property to the symbolic XRC string ID).

As this is rather "dangerous", and modifying wxWidgets' XRC library is no choice, directly accessing XRC files is the goal. This in turn necessitates:

1. Locating all potentially used XRC files.
2. Reading them in by means of an XML parser.
3. Providing appropriate accessor and query functionality to the parsed GUI description data.

Firstly, swConfigManager class, an application-global Singleton configuration abstraction makes locating XRC resource files which are recognisable by the *.xrc extension possible:

```
wxString resDir;
if (!sw::swConfigManager::GetInstance()->GetResourceDir(resDir)) {
  ...

// parse "resDir\*.xrc" files
```

Secondly, gathering the hierarchy of wxWidgets descriptively designed GUI control components employs again the highly accepted open-source XML parser Xerces-C++ based on XRC's definition [wxW05d]:

> The root node of XRC document must be <resource>. [...] The <object> node represents a single object (GUI element) and it usually maps directly to a wxWindows class instance. It has three properties: "name", "class" and "subclass". "class" must always be present, it tells XRC what wxWindows object should be created in this place. The other two are optional. "name" is ID used to identify the object. It is the value passed to the XRCID() macro and is also used to construct wxWindow's id and name attributes and must be unique among all children of the nearest container object (wxDialog, wxFrame, wxPanel, wxNotebook) upside from the object in XML nodes hiearchy (two distinct containers may contain objects with same "name", though).

So, a data structure representing the following situation must be created:

$$\begin{aligned} resDir/*.xrc &\Rightarrow m\times <resource> \\ 1\times xrc &\Rightarrow \underbrace{1\times <resource>}_{n\times<object>,\ \text{often } n=1} \end{aligned}$$

That is, one XRC resource file contains one <resource> root element which in turn can contain several <object> child elements; oftentimes it contains just one as most containers are in their own XRC file as aforementioned.

Figure 1.13 shows the developed simplified GUI control hierarchy handling and XRC parsing data structure class diagram. Navigation through the XRC hierarchy is made easy by providing access from child nodes to parent nodes and vice versa.

Finally, by accident wxDesigner's standard XRC file output revealed a bug and, not really a weakness, but rather imperfection on the part of Xerces-C++: Probably due to wxDesigner being developed by a German its generated XML output (i.e. XRC files) uses $<?xmlversion = "1.0" encoding = "ISO-8859-15"?>$ to support German umlauts by default. This Unicode encoding is not support by the XML parser, but even worse, setting an dedicated error handler does not report the unknown encoding either.

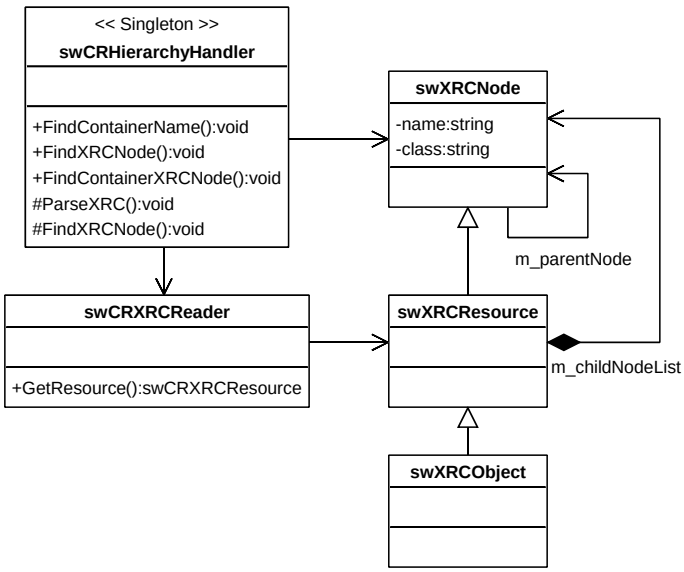


Figure 1.13: Class diagram: GUI control hierarchy handling and XRC hierarchy.

Upgrading from Xerces-C++ version 2.5 to 2.6 was not the solution, although it correctly throws a meaningful exception at least. Ways to circumvent the tool's imperfection comprise for instance switching to IBM's XML4C parser, which is based upon Xerces-C++, and supports almost all known Unicode encodings. Another idea is extending the tool by writing a so-called pluggable transcoder. Eventually, the pragmatic method of manually changing the encoding to the well supported and quasi-standard "$ISO-8859-1$" was chosen.

1.5.6.5 Overall Capture & Replay Solution

As already pointed out the capturing's core functionality is encapsulated in and orchestrated by swCREventCaptureManager class. Broadly speaking, it must process the events delegated to it by swWxGuiTestApp instance with the goal to emit code to the correct test case backup file location for simulating these events by replay.

Emitting the code immediately reduces not only the code complexity, but also the risk of loosing information in the case of occurring exceptions or crashes due to the iterative and incremental development style supporting feature by feature. No chance for optimisations and the need for little subsequent amendments (e.g. adding necessary include statements at the beginning of the test case file) are the drawbacks.

The overall C&R class diagram in Figure 1.14 indicates the currently only aimed at C++ based test case file format by swCRCapturedEvent::EmitCpp() method and swCRCppEmitter class. Nevertheless, the existing infrastructure allows a direct and easy porting to support an XML based format in the long run.

This test case format strategy results in two main objectives for the C++ code emitting, besides

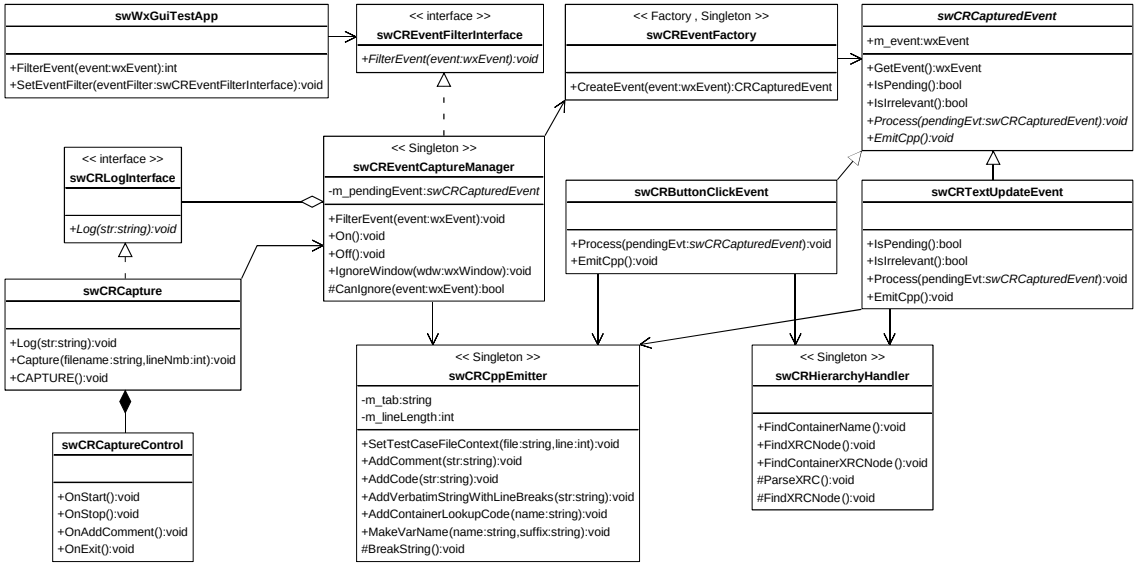


Figure 1.14: Class diagram: Overall capture & replay solution.

obviously generating correct C++ code: maintainability and readability. This is reached by a range of code emitting rules and conventions:

- Make correct variable names: first letter lowercase and alphabetical, remaining characters alphanumerical or '_'.
- Prevent multiple definitions of variables due to using generic names, e.g. wxWindow *container = ...; using a map of used variables for each event simulation code emitting.
- Only emit code for looking up event object container once using another map for container code emittings.
- Emit comment for unknown or unsupported events, so the developer can manually add the corresponding code subsequently.
- Allow configurable tabulator size.
- Break lines after (configurable) 80 characters.
- Differentiate between XRC resources and GUI resources created programmatically; only in the first case XRCID() macro may be used for example.
- Do not break lines within string constants, e.g. "...too long line"; possible options are:

  1. "...too long \
     /* some indentation */ line"
  2. "...too long "
     /* some indentation */ "line".

Only the second line splicing option guarantees the unmodified string literal value by not containing the indentation.

Both, the actual C++ test case code to simulate the captured event, as well as the information required for future modification tolerant regression testing, depends on the respective event type. For example (assuming that the XRC system creates controls by setting their name property to the symbolic XRC string ID by default: wxString xrcStrID = event.GetEventObject()->GetName(); the concrete numeric ID is, of course, int id = event.GetId(); double checking assertion id == XRCID(xrcStrID)):

- *Button clicks:*
  Identified by: event.GetEventType() == wxEVT_COMMAND_BUTTON_CLICKED
  Event simulation requires: button ID and window (button itself, or any button parent window).
  Captured button click events provide:
  - event.GetId() == button ID
  - event.GetEventObject() → pointer to button

  Symbolic identification: button->GetName() method → symbolic window ID to be used for gathering button ID and window for replay.

- *Menu item selections:*
  Identified by: event.GetEventType() == wxEVT_COMMAND_MENU_SELECTED
  Event simulation requires: menu item ID and menu parent frame (e.g. main frame via frame factory).
  Captured menu item selection events provide:
  - event.GetId() == menu item ID
  - event.GetEventObject() → pointer to frame

  Symbolic identification: menu item ID → label of menu and menu item to be used for gathering menu item ID and menu parent frame.

- *Pop-up menu item selections:*
  Identified again by: event.GetEventType() == wxEVT_COMMAND_MENU_SELECTED
  Event simulation requires: menu item ID and control (event handler) popping up the menu (e.g. tree control).
  Captured menu item selection events provide:
  - event.GetId() == menu item ID
  - event.GetEventObject() → pointer to menu; examining the type of the returned object allows to distinguish pop-up menu item selection from the normal one which returns a pointer to frame as aforementioned

Symbolic identification: menu + menu item ID → label of menu item (cf. normal menu item selections); as the pop-up menu must have been created via swWxGuiTestHelper::PopupMenu() method to allow testing, another swWxGuiTestHelper class helper method returns the control event handler.

Because of these fundamental differences a hierarchy of swCRCapturedEvent classes carries out the event type specific activities according to the Strategy pattern. Creating the correct strategy event instance depends on the concrete event type and is encapsulated in swCREventFactory class following the Factory pattern. The instantiation includes handing over the cloned wxEvent instance providing the maximum context information. Once more detecting missing clone operations for a few event types is only possible by wxWidgets' open-source character.

Among the currently supported event types some require special attention:

- Clicking toolbar tools creates events with the same type as menu item selections. Both, the event object properties and the necessary code emitting for event simulation are quite different. There is also a distinction between standard event IDs (e.g. file opening) and application specific ones.
- Changing the active page of notebook controls always creates two identical events.
- Moving the slider controls handle creates a range of different events, and at least one of them is platform dependent.
- Tree control interactions lead to a range of different events, and due to the dependency on a certain tree item expansion state several events may be created due to a single mouse click.
- Changing spin control values not only creates several events, but even different ones depending on (a) the kind of interaction (i.e. using spin buttons vs. entering numbers) and (b) the type of spin control (i.e. standard integer-typed vs. non-standard double-typed). Furthermore spin controls create text update events as used by text controls as well.

This duplicate usage of text update events is only partly to blame for the relative complexity of swCRCapturedEvent class' interface. Checking for the event object type (i.e. is it a spin control?) in swCRTextUpdateEvent::Process() method in collaboration with swCRCapturedEvent::IsIrrelevant() method allows resolving this conflict easily.

Far more involved is the solution for the repetitive text update event creation problem in case of filling out text controls: each single key stroke is creating this event. (The same applies to moving slider control or scrollbar handles.) Thus, in general all events should be collected and emitted at once to prevent emitting repetitive event simulations. Even though this contradicts the previously favoured immediate code emitting, this is still neater than continuing to emit immediately, but overwriting the current capture file for each subsequent event.

Introducing the concept of so-called pending events tackles the event collecting problem. This substantiates the decision for a two-way code emitting: (1) processing event and gathering required context dependent information, (2) actual code emitting. In between executing these steps the orchestrating CREventCaptureManager class reacts on the existence of irrelevant and pending events as shown in Listing 1.17.

While the duplicate usage of text update events as such can be easily resolved as aforementioned, the concept of pending events adds some additional complexity to this otherwise straightforward case. The implementation therefore hands over the currently pending event to the swCRCapturedEvent::Process() method with the responsibility for the pending event: It will be emitted and subsequently destroyed right after returning from this method. Thus, if a pending event should not be emitted, because it is part of the event collection assembled in this new event, it must be deleted within this method. Eventually pending event's nature requires to explicitly emit them in case of premature endings. Section A.1.7 in the appendix exemplarily lists the code for handling text update events.

Finally, in order to make capturing as easy as possible the corresponding swCRCapture class provides the aforementioned convenient CAPTURE macro (see Listing 1.18) combining everything.

1.5.6.6 Supporting VTK Render Window Interaction

The developed tools and the application ScanCAD are not just heavily GUI based, in particular ScanCAD's 3D visualisation is equally critical. Thus, besides C&R of normal GUI interactions the visualisation interactions represented by the so-called wxVTK "bridge" (cf. Section **??**) should be covered as well to some extent.

Objectives of VTK render window interaction capturing are analogous to the previously described "normal" C&R :

1. Gather initial camera and view settings (e.g. zoom); camera resetting is neither sufficient, nor a general solution.
2. Record user interaction (i.e. mouse clicks, mouse movement, keyboard input).

Subsequent replaying consists of the complementary steps:

1. Initialise camera and view with captured settings.
2. Re-execute captured interactions by replaying the recorded "script".

This camera and view dependency (to be equal in both phases) are an admittedly ugly result of the currently rather rudimentary support for VTK interaction recording. In place of an expensive actor-object-oriented layer interaction recording is based on crude mouse pointer coordinates.

```
void swCREventCaptureManager::FilterEvent (wxEvent &event) {
  if (this->IsOn ()) {
    // Check if event stems from capture dialog:
    if (this->CanIgnore (event)) {
      return;
    }
    // Check if event has already been processed:
    if ((m_event != NULL) && (&event == m_event)) {
      return;
    }
    m_event = &event;

    swCRCapturedEvent *captureEvt = swCREventFactory::GetInstance ()->
            CreateEvent (event);
    if (captureEvt) {
      captureEvt->Process (&m_pendingEvent);
      // Delete old pending event, if existing:
      if (m_pendingEvent != NULL) {
        m_pendingEvent->EmitCpp ();
        delete m_pendingEvent;
        m_pendingEvent = NULL;
      }
      if (captureEvt->IsIrrelevant ()) {
        delete captureEvt;
      } else if (captureEvt->IsPending ()) {
        m_pendingEvent = captureEvt;
      } else {
        captureEvt->EmitCpp ();
        delete captureEvt;
      }
    } else {
      // Log unsupported event details
    }
}
```

Listing 1.17: Event capturing orchestration.

Once again, the test script format question comes up, and again supporting XML in the long run is the answer. In the meantime however, utilising the current C++ test case file format is the ultimate goal. Thus, manually adding checkpoints after the capturing phase, as well as their automatic evaluation as part of replaying takes place as before.

Likewise, controlling the capturing of VTK render window interactions by means of a dialog is anew important: see Figure 1.15. Obviously there is some common capture dialog behaviour which should be reused, instead of displaying a second independent capture dialog.

This reuse raises another issue: In order to avoid any compile- and link-time dependencies on VTK for normal capturing, VTK interaction recording must become an extra target (swVtkWxGuiTesting) whilst still reusing standard capturing classes (if required). Consequently, another helper class called swVtkWxGuiTestHelper in analogy to swWxGuiTestHelper class is necessary as depicted in the class diagram in Figure 1.16.

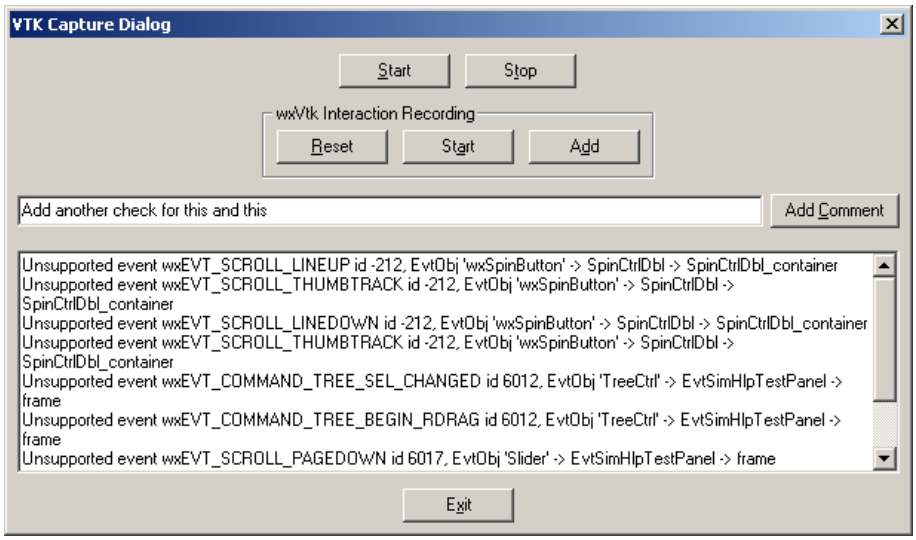


Figure 1.15: VTK interaction Capture dialog.

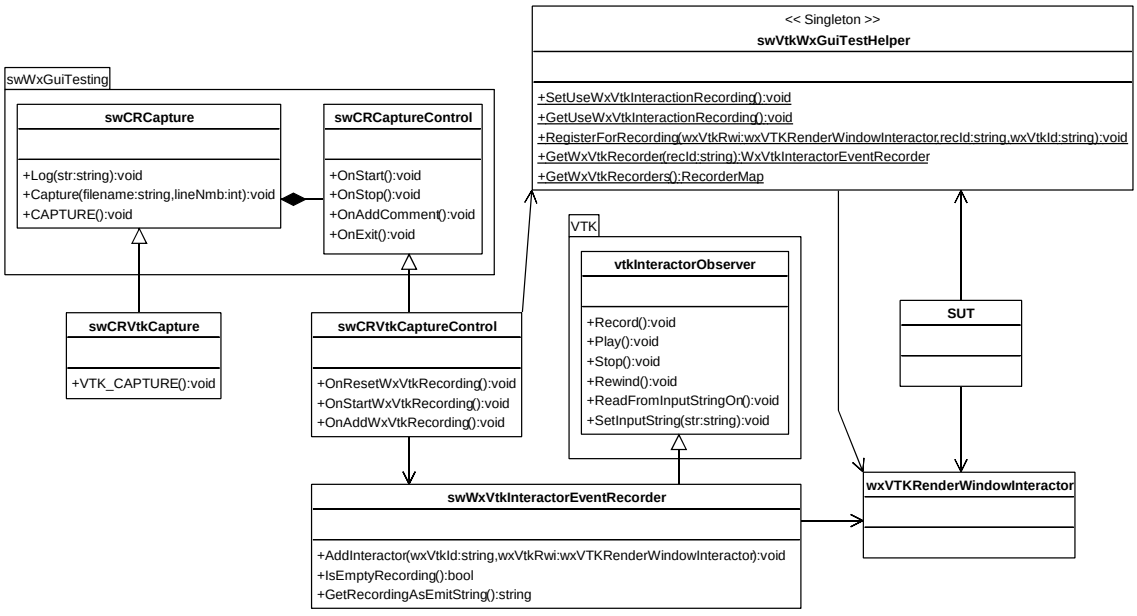


Figure 1.16: Class diagram: VTK interaction capture & replay solution.

```
#define CAPTURE {                                                              \
  wxApp *app = wxTheApp;                                                       \
  wxASSERT (app != NULL);                                                      \
  swTst::swWxGuiTestApp *guiTestApp = dynamic_cast< swTst::swWxGuiTestApp * >(
        app); \
  wxASSERT (guiTestApp != NULL);                                               \
  guiTestApp->SetEventFilter (swTst::swCREventCaptureManager::GetInstance ())
        ; \
                                                                               \
  wxString excMsg;                                                             \
  swTst::swCRCapture *capture = new swTst::swCRCapture ();                     \
  try {                                                                        \
      capture->Capture (__FILE__, __LINE__);                                   \
  } catch (std::exception &e) {                                                \
      excMsg = e.what ();                                                      \
  } catch (...) {                                                              \
      excMsg = "Unexpected capturing exception";                               \
  }                                                                            \
  guiTestApp->SetEventFilter (NULL);                                           \
  delete capture;                                                              \
  swTst::swCRCppEmitter::Destroy ();                                           \
  if (!excMsg.IsEmpty ()) {                                                    \
    CPPUNIT_FAIL (excMsg.c_str ());                                            \
  }                                                                            \
}
```

Listing 1.18: Capturing macro.

In the hope of coming up with a replacement for the coordinate-oriented recording storing and restoring render window and camera settings is also of rather temporary nature: take sizes and positions of wxWindow class containers starting from the render window up to and including the root top-level container into account.

The actual recording and replaying of interactions is heavily based on vtkInteractionEventRecorder class from VTK, which is in fact already used for some rudimentary testing of VTK's widgets. Besides a few important helper functionalities, it lacks one vital feature: Several VTK render windows (e.g. the so-called MultiView containing one perspective 3D view and several parallel or slice 2D views) may be part of one wxWidget container. As the user most likely wants to interact with not just one of the render windows, all interactions must be placed into the same recording context. Thus, in opposition to the standard VTK class which serves one single render window interactor, swWxVtkInteractionEventRecorder class must support a kind of multiplexing.

The underlying functionality from vtkInteractionEventRecorder class is also the reason for the different capture dialog interface consisting of "Reset", "Start" and "Add" buttons in contrast to normal capturing's solely "Start" and "Stop" buttons: interactions are stored as simple lists describing coordinates and interaction types in files. So, resetting deletes the file with captured interactions, while starting obviously switches on the interaction recorders, and adding first switches the recorders off before emitting the list of interactions in the file as one single event simulation block.

Keeping the interaction recording integration as transparent and non-intrusive as possible for the SUT (e.g. ScanCAD) is important. ScanCAD, for example, instantiates the wxVTKRenderWindowInteractor class (abbreviated as wxVtkRwi in the following) like:

```
rwi = new wxVTKRenderWindowInteractor (parent, -1);
rwi->UseCaptureMouseOn ();
```

The developed solution provides a helper method to register this wxVtkRwi instance for recording. As already pointed out several views must be able to use the same interaction recorder, and thus, must employ the same identifier during registration:

```
swTst::swVtkWxGuiTestHelper::RegisterForRecording (rwi, "MyRecorderIdentifier", "
        MyWxVtkRwiIdentifier");
```

This registration is defined as:

```
swVtkWxGuiTestHelper::RegisterForRecording (...) {
  swVtkInteractorEventRecorder * recorder = NULL;
  recorder = this->GetWxVtkRecorder (MyRecorderIdentifier);
  if (recorder == NULL) {
    recorder = swVtkInteractorEventRecorder::New ();
    recorder->SetFileName (MyRecorderIdentifier << "_wxVtk.rec");
    this->RegisterRecorder (MyRecorderIdentifier);
  }
  wxASSERT (recorder != NULL);
  recorder->AddInteractor (rwi, MyWxVtkRwiIdentifier);
}
```

In order to allow identifying a single wxVtkRwi – even if it is part of a so-called MultiView (cf. multiplexing) – for replay, the render window identifier is appended to the interaction recording (default format: EventName X Y Ctrl Shift Keycode RepeatCount KeySymbol) by swWxVtkInteractionEventRecorder class:

```
EnterEvent 46 241 0 0 0 0 i MyWxVtkRwiIdentifier
```

Thus, the Play() method has to query the corresponding wxVtkRwi first, before invoking the event:

```
wxVtkRwi->SetEventPosition (pos);
wxVtkRwi->SetControlKey (ctrlKey);
wxVtkRwi->SetShiftKey (shiftKey);
wxVtkRwi->SetKeyCode (keyCode);
wxVtkRwi->SetRepeatCount (repeatCount);
wxVtkRwi->SetKeySym (keySym);

wxVtkRwi->InvokeEvent (ievent, NULL);
```

Finally, swCRVtkCapture class provides also a convenient VTK_CAPTURE macro, which is identical with the CAPTURE macro with the exception of the concrete capturing class name.

Concluding, C&R of VTK render window interactions requires only moderate development effort by reusing a range of existing concepts and implementations. Nevertheless, this approach being based on crude mouse pointer coordinates may lead to test scripts which are rather poor and difficult to maintain. However, in consideration of the extremely expensive actor-object-oriented layer (handling each standard and non-standard VTK widget and interactor) for interaction recording this seems like a passable pragmatic solution. For the purpose of better maintainability a combination of using the coordinate based C&R with a "manual" actor-object-oriented coordinate calculation applied on the same interaction event recorder appears to be the best compromise (see Section A.1.8 in the appendix for an example).

1.6 Conclusion

Automated testing is a vital part in demonstrating the robustness and correctness of ScanCAD's source code which will be the basis for a future commercial product; and this in turn is an important precondition to allow for evolvability and maintenability. The development of the testing environment and framework presented in this thesis proved invaluable in gaining and maintaining confidence in the quality of the code, whilst fulfilling all initial requirements with complete satisfaction. While automated testing is accepted practice now, the proposed approach has unique advantages:

- Not only are all testing levels supported, i.e. (a) normal unit tests of core algorithms or functionality, (b) integration or GUI tests of isolated parts of an application, and (c) acceptance or application/system tests involving the whole system; but the boundaries between them are transparent from the test case developer's perspective. That is, they bear great resemblance to each other and all differences, as well as the few unfamiliar and non-standard issues, are either automatically generated or hidden:
  - wxCRP templates generate test skeletons.
  - Events are simulated by simple calls to the EvenSimulationHelper class.
  - C&R functionality generates any GUI interaction, i.e. code for event simulation, within the test case skeletons; (including VTK 3D render window interaction).

  Thus, the test case developer must only insert the actual checkpoints and assertions anymore.
- The underlying pragmatism of the approach is reflected by (a) the relatively small size of the testing framework considering the extent of its functionality, i.e. again increasing evolvability and maintenability; commercial GUI testing tools typically consist of numerous man years of work and therefore much more complexity; (b) the basic idea is generalisable to other GUI toolkits and programming languages; and (c) some restrictions based on decisions that obviously lead to minor imperfections as stated below.

- Even the GUI testing is in theory platform independent because it builds upon wxWidgets, but does not require changing wxWidgets internally, and only uses parts which supposedly are available on all (major) platforms. By comparison, there are quite a number of GUI testing tools that are not cross platform; and if they are, at least their core implementation is per definition always fundamentally different from platform to platform. Hence, this is another indicator for high maintenability and evolvability of this solution. In practice, the testing framework's successful operation was demonstrated on different Microsoft Windows platforms, while raising only very few minor and solvable issues on Linux (Red Hat distribution): it seems as if certain GUI controls necessitate additional idle events for GTK2 subsequent to simulating events.

- The ease of use (i.e. test case developer-friendliness) is impressive, because the test cases can be implemented using the same programming language (`C++`), same techniques and libraries (wxWidgets) as used for developing the system/application under test:

  - GUI test cases principally fit into the "normal" testing framework (CppUnit) – again something with which the test case developer is already familiar with.

  - Commercial GUI testing tools not only often employ proprietary scripting languages for the test cases, but in addition the scripting language is (almost) without exception different to the language used for writing the system/application under test. The approach presented here does not carry the overhead of having to learn an extra (and mostly proprietary) scripting language, and allows an easy debugging in case of failing assertions due to the close integration.

  - Moreover, test cases consist of standard wxWidgets and `C++` code with only a few specific new parts (e.g. event simulation), but all of them are either hidden in "convenience" code, or automatically generated (cf. Capture & Replay).

  - Maintenance problems caused by some short-sighted Capture & Replay tools (e.g. screen or window size dependence, based on coordinates only; using screenshots for check points) are prevented by being GUI-object/control oriented. (This does not apply to Capture & Replay of VTK 3D window interaction though.)

- Available Capture & Replay functionality like commercial GUI testing tools increases the test case development/writing productivity significantly: user interactions with the GUI/application under test are recorded (captured) and the corresponding test case (in the same format) for automatic replay simultaneously created. Thus, as already indicated, even the few specific new parts (i.e. event simulation) which may be quite time consuming to do otherwise, are done automatically. Due to the applied pragmatism there are some imperfections: Capture & Replay does not create all necessary include statements, and sometimes the generated test cases can be written more straightforwardly and more cleanly.

- Of great importance is the fact that there is no need to modify wxWidgets internally. However, this in turn demands that the system/application under test must be adapted in a few locations to support this pragmatic approach; such modifications are never necessary for advanced commercial GUI testing tools.

Any of these items also shows the fulfilment of one or more quality goals like maintenability, evolvability, understandability, learnability and familiarity.

Test-Driven Development Broadly speaking, "pure" Test-Driven Development as described by Kent Beck in [Bec02] was not realisable for the author, with the exception of rather simple and straightforward cases. Of course, drawing the line between simple and complex is subjective and depends mostly on experience. Ignoring this metaphysical question, (1) creating a simple design according to agile methodologies, (2) implementing this with unit tests – sometimes even after writing the actual functionality, i.e. no test first style –, and (3) evolving the design subsequently was found easier to realise.

Nonetheless, even forcing Test-Driven Development with the aforementioned variation has proven to be a real success. So, not only slips of the pen like forgetting the actual make-directory call in a class that represents temporary directories, but also mistakes that may have resulted in long hours of searching and debugging (e.g. implementation mistake in copy & assignment constructor of an essential ScanCAD class) were found and solved in minutes.

Further Work By default, failing assertions (of course only in debug mode) or (throwing) exceptions stop the test execution immediately at the error location. This is desired behaviour during the development of test cases, where the test execution is started in debug mode, and breaking test cases due to the aforementioned reasons allow a quick diagnosis based on stack trace, register values, etc. at the error location. However, in release mode a crash of the test execution impedes any test result output.

So, Continuous Integration (= automatic regression testing) – that is carried out on a server without any human interaction whatsoever – is doomed to fail in such situations. Here the solution is to detect the test case failure as soon as possible and to handle it appropriately. Unfortunately, sometimes closing the system/application under test gracefully fails, and so this strategy results in infinite main message loops, that is, daemon processes that will not produce any test result output anymore. The source of the problem seems to lie in the event processing interruption. Therefore, test cases are currently not always aborted at the first possible chance, i.e. failure detection; instead, after notifying the testing framework of the test case failure, it is tried to complete the processing of current events which can lead to a continuation of the already failed test case. Consequently, this solution is not at all perfect, but has shown to be sufficiently pragmatic in practice.

Another potential solution could be the modification of wxWidgets to prevent the event processing

interruption by catching all exceptions. This should allow the detection of test case failures as soon as possible without the aforementioned troubles (no graceful closing due to infinite loop). Nonetheless, modifying wxWidgets may lead to a maintenance nightmare in the likely case of updating the wxWidgets version regularly.

Probably the best solution would also address the different, but even more dramatic problem of abnormal terminations (= crashes) due to the close integration of the testing framework with the system/application under test. For instance, while in debug mode assertions for finding a required GUI control can fail without consequences (the assertion is stopping the test before this GUI control is accessed), an access violation is the result in release mode. Thus, the *decoupling* of testing framework and system/application under test is critical for Continuous Integration. Taking the aforementioned discussion into account the decoupling can be unfavourable during test case development, so the ultimate goal is to support both methods.

One way of the decoupling is running each test case in a second (= child) process that is started by the test driver from the main (= parent) process. Interestingly, CppUnit2 [Lep05b] – the successor of CppUnit, that is the current base of the testing framework – aims at providing exactly this approach [Lep05a]: "In the case of [an] infinite loop it terminates the child process. If either a crash or an infinite loop [has] occurred, a new child process is spawned and testing resume[s] after the problematic test case."

A second way is using a different build approach to generate a single and stand-alone executable for each test case. This is to some extent similar to the testing system provided by CMake and CTest [Kit05]. This raises the question if the testing framework should converge to CMake and CTest, since CMake is already used for the cross platform build system. Both decoupling ways have – in addition to the fact that crashes of a single test case do not matter anymore – also the advantage that the memory is always freed after each test case. While this is also linked with crashes, its importance became also apparent in the course of developing and testing ScanCAD where memory leaks involving huge 3D image structures can have quite negative consequences (e.g. slowing down due to need of swapping to disk, or theoretically even out of memory exceptions).

Another, but currently less critical item for further work is improving the support of VTK 3D render window interaction by C&R. As already pointed out, the present approach being based on crude mouse pointer coordinates may lead to test scripts which are poor and difficult to maintain. On the other hand, creating an actor-object-oriented layer – that handles each standard and non-standard VTK widget and interactor – is too much effort in consideration of the expected benefit: test cases involving the VTK 3D render window interaction are the minority. Nevertheless adding generic and basic helper functionality like a method that simulates a left mouse button click on a certain coordinate can be a first step towards better maintainable test scripts. This helper functionality can than be combined with using the coordinate based C&R and a "manual" actor-object-oriented coordinate calculation as a compromise.

Finally, saving testing protocols with configurable detail could be another helpful feature facilitating the finding and solving of test case failures.

Appendix A

Design & Implementation Artefacts

For the sake of comprehension some of the most important design diagrams and pieces of code are presented within this chapter, usually without extra explanations.

A.1 Automated Testing

Without doubt the actual implementation, i.e. the source code, is in the centre of automated testing. This applies on the automated testing framework, as well as its utilisation. Therefore the number of code listings and other development artefacts in this section is comparatively high.

A.1.1 CppUnit Test Driver

Generating CppUnit test drivers by means of wxCRP allows the user to choose file name and testing mode (standard unit test vs. GUI unit test vs. application unit test). Listing A.1 shows the output for default settings using standard unit tests.

A.1.2 CppUnit Test Class

Listings A.2 and A.3 present wxCRP's generated default test case class output. The user can select class name, namespace and initial skeleton methods (constructor, destructor; test fixture setUp() and tearDown() methods; sample test case).

```
#ifndef MYCPPUNITTESTCLASS_H
#define MYCPPUNITTESTCLASS_H

#ifdef __GNUG__
```

```
#include <cppunit/CompilerOutputter.h>
#include <cppunit/extensions/TestFactoryRegistry.h>
#include <cppunit/ui/text/TestRunner.h>

int main (int argc, char* argv[])
{
  // Get the top level suite from the registry
  CPPUNIT_NS::Test *suite = CPPUNIT_NS::TestFactoryRegistry::getRegistry().makeTest
        ();

  // Adds the test to the list of test to run
  CPPUNIT_NS::TextUi::TestRunner runner;
  runner.addTest (suite);

  // Change the default outputter to a compiler error format outputter
  runner.setOutputter (new CPPUNIT_NS::CompilerOutputter (&runner.result (),
      std::cerr));

  // Run the test - don't close window in debug mode at the end
  #if (defined (__WXDEBUG__) || defined (_DEBUG)) && !defined (NDEBUG)
    bool wasSucessful = runner.run ("All Tests", true);
  #else
    bool wasSucessful = runner.run ();
  #endif

  // Return error code 1 if the one of test failed.
  return wasSucessful ? 0 : 1;
}
```

Listing A.1: CppUnit test driver generated by wxCRP.

```
  #pragma interface "MyCppUnitTestClass.h"
#endif

#include <cppunit/extensions/HelperMacros.h>

namespace swTst {


/*! \class MyCppUnitTestClass
    \brief ...

    ...
*/
class MyCppUnitTestClass : public CPPUNIT_NS::TestFixture {
  CPPUNIT_TEST_SUITE( MyCppUnitTestClass );

    CPPUNIT_TEST( testExample );

  CPPUNIT_TEST_SUITE_END();

public:
  /*! \fn virtual void setUp ()
      \brief Set up context before running a test.

      ...
  */
  virtual void setUp ();


  /*! \fn virtual void tearDown ()
      \brief Clean up after the test run.

      ...
  */
```

```
  virtual void tearDown ();


  /*! \fn virtual void testExample ()
      \brief Example test case.

      ...
  */
  virtual void testExample ();


protected:

private:

};

} // End namespace swTst

#endif // MYCPPUNITTESTCLASS_H
```

Listing A.2: CppUnit test class header file generated by wxCRP.

```
#ifdef __GNUG__
  #pragma implementation "MyCppUnitTestClass.h"
#endif

#include "MyCppUnitTestClass.h"

namespace swTst {


// Register test suite:
CPPUNIT_TEST_SUITE_REGISTRATION( MyCppUnitTestClass );


void MyCppUnitTestClass::setUp ()
{
}


void MyCppUnitTestClass::tearDown ()
{
}


void MyCppUnitTestClass::testExample ()
{
}

} // End namespace swTst
```

Listing A.3: CppUnit test class implementation file generated by wxCRP.

A.1.3 Unit Testing Example

Listing A.4 is a unit testing example for the purpose of testing the saving of the currently active document in the ScanCAD specific project file format. Due to application dependencies in the employed code (cf. Section 1.3.2) first an application stub is instantiated in the setUp() method.

The test case itself consists of the following steps and checks:

1. Load a ScanCAD project test file.
2. Check if loading succeeded.
3. Save the loaded document immediately in a temporary directory in the shape of another ScanCAD project file.
4. Check if saving succeeded.
5. Check if sizes of original file and re-saved file are the same.

Of course, this test case is not sufficient to show that saving project files works. Another test case class testing the correct loading of project files is equally important to either (a) more checks of individual project file entries in the depicted testSaveSCPFile() test case method, or (b) putting these additional checks in another, separate test case method.

```
void sccSCPFileSaverTest::setUp ()
{
  sw::swPseudoApp::SetPseudoAppInstance ();
}


void sccSCPFileSaverTest::tearDown ()
{
  // Nothing to do
}


void sccSCPFileSaverTest::testSaveSCPFile ()
{
  wxString dir = "../../TestData/scpv10/";
  wxString filename = dir + "test.scp";

  scc::sccDocument doc;
  // Make it the active document, needed because during loading view updates
  // are carried out based on active document:
  swActiveDocument::GetInstance ()->SetActiveDocument (&doc);

  scc::sccSCPFileLoader loader;
  CPPUNIT_ASSERT_MESSAGE ("Opening failed", loader.LoadDocument (filename, &doc));

  // Re-save project file:
  sw::swTempDir tempDir;
  tempDir.SetDirnamePrefix ("testSaveSCPFile");
  tempDir.MakeDir ();

  wxString savedSCPFilename = tempDir.GetDirname () + "test.scp";

  scc::sccSCPFileSaver saver;
  CPPUNIT_ASSERT_MESSAGE ("Saving failed", saver.SaveDocument (savedSCPFilename,
          doc));

  CPPUNIT_ASSERT_MESSAGE (wxString::Format ("Saving could not create file: %s",
          savedSCPFilename).c_str (), true == wxFileName::FileExists (
          savedSCPFilename));

  wxFile loadedSCPFile, savedSCPFile;
  loadedSCPFile.Open (filename);
  savedSCPFile.Open (savedSCPFilename);

  CPPUNIT_ASSERT_MESSAGE (wxString ("Saving corrupted file").c_str (), savedSCPFile
          .Length () == loadedSCPFile.Length ());
```

```
  loadedSCPFile.Close ();
  savedSCPFile.Close ();

  // Un-activate document:
  swActiveDocument::GetInstance ()->SetActiveDocument (NULL);
}
```

Listing A.4: Unit testing example: testing project file saving.

A.1.4 Protocol of Applying Test-Driven Development

For the sake of completeness a protocol of applying TDD (see Section 1.4) on the development of a rather moderate feature consisting of two classes is shown subsequently. It starts with the initial files shown as a whole, ending up with the most recent versions using only incremental changes introduced with comments. Each little (cf. time stamps of files) development step concludes with running CppUnit tests, followed by a line separation like "`//##########...`".

The actually implemented feature is part of automated application testing and demonstrates a nice style of recursion by developing parts of the testing framework based on tests: A registry managing provoked warnings provides the required functionality of discriminating between provoked (initiated by test cases) and unexpected warnings (= real test case failures). For more details on the objectives of this feature, see Section 1.5.4.2.

Here is the development protocol of the feature's TDD:

```
swWxGuiTestProvokedWarningRegistryTest.h 1.1, 2004/12/20 15:4:13, Add provoked warning
        handling (by means of registry) plus CppUnit tests for its TDD
swWxGuiTestProvokedWarningRegistryTest.cpp 1.1, 2004/12/20 15:4:13, Add provoked warning
        handling (by means of registry) plus CppUnit tests for its TDD
swWxGuiTestProvokedWarning.h   1.1, 2004/12/20 15:4:13, Add provoked warning handling (
        by means of registry) plus CppUnit tests for its TDD
swWxGuiTestProvokedWarning.cpp 1.1, 2004/12/20 15:4:13, Add provoked warning handling (
        by means of registry) plus CppUnit tests for its TDD
swWxGuiTestProvokedWarningRegistry.h 1.1, 2004/12/20 15:4:13, Add provoked warning
        handling (by means of registry) plus CppUnit tests for its TDD
swWxGuiTestProvokedWarningRegistry.cpp 1.1, 2004/12/20 15:4:13, Add provoked warning
        handling (by means of registry) plus CppUnit tests for its TDD

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistryTest.h,v
// Original Author/Owner: reinhold
//
// Revision: 1.2 Name: State: Exp
//
// Date: 2004/12/2016:42:10 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

#ifndef SWWXGUITESTPROVOKEDWARNINGREGISTRYTEST_H
#define SWWXGUITESTPROVOKEDWARNINGREGISTRYTEST_H

#ifdef __GNUG__
    #pragma interface "swWxGuiTestProvokedWarningRegistryTest.h"
#endif

#include <cppunit/extensions/HelperMacros.h>

namespace swTst {


/*! \class swWxGuiTestProvokedWarningRegistryTest
    \brief Test WxGuiTestProvokedWarningRegistry.
```

```
*/
class swWxGuiTestProvokedWarningRegistryTest : public CPPUNIT_NS::TestFixture
{
    CPPUNIT_TEST_SUITE( swWxGuiTestProvokedWarningRegistryTest );

        CPPUNIT_TEST( testUnRegister );
        CPPUNIT_TEST( testTimeout );
        CPPUNIT_TEST( testDetection );

    CPPUNIT_TEST_SUITE_END();

public:
    /*! \fn virtual void setUp ()
        \brief Set up context before running a test.
    */
    virtual void setUp ();


    /*! \fn virtual void tearDown ()
        \brief Clean up after the test run.
    */
    virtual void tearDown ();


    /*! \fn virtual void testUnRegister ()
        \brief Test case for simple registration and unregistration.
    */
    virtual void testUnRegister ();


    /*! \fn virtual void testTimeout ()
        \brief Test case for timeout exceeding.
    */
    virtual void testTimeout ();


    /*! \fn virtual void testDetection ()
        \brief Test case for warning detection.
    */
    virtual void testDetection ();

protected:

private:

};

} // End namespace swTst

#endif // SWWXGUITESTPROVOKEDWARNINGREGISTRYTEST_H

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistryTest.cpp,v
// Original Author/Owner: reinhold
//
// Revision: 1.1 Name: State: Exp
//
// Date: 2004/12/2015:04:13 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

#ifdef __GNUG__
    #pragma implementation "swWxGuiTestProvokedWarningRegistryTest.h"
#endif

#include "swWxGuiTestProvokedWarningRegistryTest.h"

#include "swWxGuiTestProvokedWarningRegistry.h"

namespace swTst {


// Register test suite:
CPPUNIT_TEST_SUITE_REGISTRATION( swWxGuiTestProvokedWarningRegistryTest );


void swWxGuiTestProvokedWarningRegistryTest::setUp ()
{
    // Nothing to do
}
```

```
void swWxGuiTestProvokedWarningRegistryTest::tearDown ()
{
    swWxGuiTestProvokedWarningRegistry::Destroy ();
}


void swWxGuiTestProvokedWarningRegistryTest::testUnRegister ()
{
    swTst::swWxGuiTestProvokedWarning warning (
            _("Direct CAD Voxelisation Warning"), NULL, 2);
    swTst::swWxGuiTestProvokedWarningRegistry &provWarningRegistry =
            swTst::swWxGuiTestProvokedWarningRegistry::GetInstance ();
    provWarningRegistry.RegisterWarning (warning);

    CPPUNIT_ASSERT_MESSAGE ("Warning is not registered",
            provWarningRegistry.IsRegistered (warning));

    CPPUNIT_ASSERT_MESSAGE ("Warning was already detected",
            !provWarningRegistry.WasDetected (warning));

    provWarningRegistry.UnregisterWarning (warning);

    CPPUNIT_ASSERT_MESSAGE ("Warning is registered",
            !provWarningRegistry.IsRegistered (warning));

    CPPUNIT_ASSERT_MESSAGE ("Warning was already detected",
            !provWarningRegistry.WasDetected (warning));
}


void swWxGuiTestProvokedWarningRegistryTest::testTimeout ()
{
    swTst::swWxGuiTestProvokedWarning warning (
            _("Direct CAD Voxelisation Warning"), NULL, 2);
    swTst::swWxGuiTestProvokedWarningRegistry &provWarningRegistry =
            swTst::swWxGuiTestProvokedWarningRegistry::GetInstance ();
    provWarningRegistry.RegisterWarning (warning);

    CPPUNIT_ASSERT_MESSAGE ("Warning is not registered",
            provWarningRegistry.IsRegistered (warning));

    CPPUNIT_ASSERT_MESSAGE ("Warning was already detected",
            !provWarningRegistry.WasDetected (warning));

    // Kinda unregistration due to timeout exceeding:
    ::wxSleep (3);

    CPPUNIT_ASSERT_MESSAGE ("Warning is registered",
            !provWarningRegistry.IsRegistered (warning));

    CPPUNIT_ASSERT_MESSAGE ("Warning was already detected",
            !provWarningRegistry.WasDetected (warning));
}


void swWxGuiTestProvokedWarningRegistryTest::testDetection ()
{
    swTst::swWxGuiTestProvokedWarning warning (
            _("Direct CAD Voxelisation Warning"), NULL, 2);
    swTst::swWxGuiTestProvokedWarningRegistry &provWarningRegistry =
            swTst::swWxGuiTestProvokedWarningRegistry::GetInstance ();
    provWarningRegistry.RegisterWarning (warning);

    CPPUNIT_ASSERT_MESSAGE ("Warning is not registered",
            provWarningRegistry.IsRegistered (warning));

    CPPUNIT_ASSERT_MESSAGE ("Warning was already detected",
            !provWarningRegistry.WasDetected (warning));

    // Set warning to have been detected:
    provWarningRegistry.SetWarningAsDetected (warning);

    CPPUNIT_ASSERT_MESSAGE ("Warning is not registered",
            provWarningRegistry.IsRegistered (warning));

    CPPUNIT_ASSERT_MESSAGE ("Warning was not detected",
            provWarningRegistry.WasDetected (warning));
}

} // End namespace swTst
```

```
////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarning.h,v
// Original Author/Owner: reinhold
//
// Revision: 1.1 Name: State: Exp
//
// Date: 2004/12/2015:04:13 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

#ifndef SWWXGUITESTPROVOKEDWARNING_H
#define SWWXGUITESTPROVOKEDWARNING_H

#ifdef __GNUG__
    #pragma interface "swWxGuiTestProvokedWarning.h"
#endif

#include "Common.h"

#include "wx/datetime.h"

namespace swTst {


/*! \class swWxGuiTestProvokedWarning
    \brief Holds information semi-uniquely identifying provoked/expected warnings.

    Based on the specified timeout interval provoked warnings must occur, i.e.
    pop-up within a given time span. This avoids conflicts with other provoked
    warnings issued later with overlapping or even equal caption and message
    parameters (=> semi-uniquely identification).
*/
class swWxGuiTestProvokedWarning
{
public:
    /*! \fn swWxGuiTestProvokedWarning ()
        \brief Constructor

        \param caption caption of provoked/expected warnings
        \param message message of provoked/expected warnings, or NULL if only
            caption should be used for convenience reasons; message is deleted
            in destructor if none-NULL
        \param timeout timeout in seconds starting during object creation
    */
    swWxGuiTestProvokedWarning (const wxString &caption,
            const wxString *message, const unsigned int timeout);


    /*! \fn virtual ~swWxGuiTestProvokedWarning ()
        \brief Destructor
    */
    virtual ~swWxGuiTestProvokedWarning ();


    /*! \fn virtual const wxString & GetCaption () const
        \brief Get caption being part of the semi-uniquely identification.

        \return caption of expected warning
    */
    virtual const wxString & GetCaption () const;


    /*! \fn virtual const wxString * GetMessage () const
        \brief Get message being part of the semi-uniquely identification.

        If NULL, only caption should be used for convenience reasons.

        \return message of expected warning, may be NULL
    */
    virtual const wxString * GetMessage () const;


    /*! \fn virtual const wxDateTime & GetTimeout () const
        \brief Get timeout of the provoked/expected warning.

        \return timeout of expected warning
    */
    virtual const wxDateTime & GetTimeout () const;

protected:
```

```
private:
    wxString m_caption;
    const wxString *m_message;
    wxDateTime m_timeout;

private:
    // No copy and assignment constructor:
    swWxGuiTestProvokedWarning (const swWxGuiTestProvokedWarning &rhs);
    swWxGuiTestProvokedWarning & operator= (const swWxGuiTestProvokedWarning &rhs);
};

} // End namespace swTst

#endif // SWWXGUITESTPROVOKEDWARNING_H

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarning.cpp,v
// Original Author/Owner: reinhold
//
// Revision: 1.1 Name: State: Exp
//
// Date: 2004/12/2015:04:13 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

#ifdef __GNUG__
    #pragma implementation "swWxGuiTestProvokedWarning.h"
#endif

#include "swWxGuiTestProvokedWarning.h"

namespace swTst {


swWxGuiTestProvokedWarning::swWxGuiTestProvokedWarning (const wxString &caption,
        const wxString *message, const unsigned int timeout) :
m_caption (caption),
m_message (message)
{
    m_timeout.SetToCurrent ();
    m_timeout.Add (wxTimeSpan::Seconds (timeout));
}


swWxGuiTestProvokedWarning::~swWxGuiTestProvokedWarning ()
{
    if (m_message != NULL) {

        delete m_message;
    }
}


const wxString & swWxGuiTestProvokedWarning::GetCaption () const
{
    return m_caption;
}


const wxString * swWxGuiTestProvokedWarning::GetMessage () const
{
    return m_message;
}


const wxDateTime & swWxGuiTestProvokedWarning::GetTimeout () const
{
    return m_timeout;
}

} // End namespace swTst

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistry.h,v
// Original Author/Owner: reinhold
//
// Revision: 1.1 Name: State: Exp
//
// Date: 2004/12/2015:04:13 Author: reinhold
//
```

```
////////////////////////////////////////////////////////////////////////////

#ifndef SWWXGUITESTPROVOKEDWARNINGREGISTRY_H
#define SWWXGUITESTPROVOKEDWARNINGREGISTRY_H

#ifdef __GNUG__
    #pragma interface "swWxGuiTestProvokedWarningRegistry.h"
#endif

#include "Common.h"

#include <map>

#include "swWxGuiTestProvokedWarning.h"

namespace swTst {


/*! \class swWxGuiTestProvokedWarningRegistry
    \brief Manage registered provoked warnings allowing the detection of
    unexpected (real) test case failures.

    (Using Singleton pattern.)
*/
class swWxGuiTestProvokedWarningRegistry
{
public:
    /*! \fn static swWxGuiTestProvokedWarningRegistry & GetInstance ()
        \brief Get single private instance (Singleton pattern).

        \return single private registry instance
    */
    static swWxGuiTestProvokedWarningRegistry & GetInstance ();


    /*! \fn static void Destroy ()
        \brief Threadsafe destruction of static singleton instance.
    */
    static void Destroy ();


    /*! \fn virtual void RegisterWarning (const swWxGuiTestProvokedWarning &warning)
        \brief Register provoked/expected warning.

        \param warning provoked/expected warning to register
    */
    virtual void RegisterWarning (const swWxGuiTestProvokedWarning &warning);


    /*! \fn virtual void UnregisterWarning (const swWxGuiTestProvokedWarning &warning)
        \brief Unregister and remove provoked/expected warning from container.

        \param warning provoked/expected warning to unregister
    */
    virtual void UnregisterWarning (const swWxGuiTestProvokedWarning &warning);


    /*! \fn virtual bool IsRegistered (const swWxGuiTestProvokedWarning &warning) const
        \brief Check if provoked/expected warning is registered and time out
        interval has not exceeded.

        If time out inteval is exceeded, the registered warning is unregistered
        for the sake of simplicity.

        \param warning provoked/expected warning to check for registration
        \return true, if warning is registered
    */
    virtual bool IsRegistered (const swWxGuiTestProvokedWarning &warning) const;


    /*! \fn virtual bool WasDetected (const swWxGuiTestProvokedWarning &warning) const
        \brief Check if this provoked/expected warning was detected.

        \param warning provoked/expected warning to check for detection
        \return true, if warning was detected within specified time out interval
    */
    virtual bool WasDetected (const swWxGuiTestProvokedWarning &warning) const;


    /*! \fn virtual void SetWarningAsDetected (const swWxGuiTestProvokedWarning &warning)
        \brief Set provoked/expected warning to has been detected.
```

```
        \param warning detected provoked/expected warning
    */
    virtual void SetWarningAsDetected (const swWxGuiTestProvokedWarning &warning);

protected:
    /*! \fn swWxGuiTestProvokedWarningRegistry ()
        \brief Constructor
    */
    swWxGuiTestProvokedWarningRegistry ();


    /*! \fn virtual ~swWxGuiTestProvokedWarningRegistry ()
        \brief Destructor
    */
    virtual ~swWxGuiTestProvokedWarningRegistry ();

private:
    static swWxGuiTestProvokedWarningRegistry *ms_instance;
    // Key is provoked/expected warning, value if it was detected:
    typedef std::map< const swWxGuiTestProvokedWarning *, bool > ProvokedWarningMap;
    ProvokedWarningMap warnings;

private:
    // No copy and assignment constructor:
    swWxGuiTestProvokedWarningRegistry (const swWxGuiTestProvokedWarningRegistry &rhs);
    swWxGuiTestProvokedWarningRegistry & operator= (const
            swWxGuiTestProvokedWarningRegistry &rhs);
};

} // End namespace swTst

#endif // SWWXGUITESTPROVOKEDWARNINGREGISTRY_H

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistry.cpp,v
// Original Author/Owner: reinhold
//
// Revision: 1.1 Name: State: Exp
//
// Date: 2004/12/2015:04:13 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

#ifdef __GNUG__
    #pragma implementation "swWxGuiTestProvokedWarningRegistry.h"
#endif

#include "swWxGuiTestProvokedWarningRegistry.h"

namespace swTst {


// Init single instance:
swWxGuiTestProvokedWarningRegistry *swWxGuiTestProvokedWarningRegistry::ms_instance =
        NULL;


swWxGuiTestProvokedWarningRegistry::swWxGuiTestProvokedWarningRegistry ()
{
    // Nothing to do
}


swWxGuiTestProvokedWarningRegistry::~swWxGuiTestProvokedWarningRegistry ()
{
    ProvokedWarningMap::iterator it;
    for (it = warnings.begin (); it != warnings.end (); it++) {

        this->UnregisterWarning (*((*it).first));
    }
    warnings.clear ();
}


swWxGuiTestProvokedWarningRegistry & swWxGuiTestProvokedWarningRegistry::GetInstance ()
{
    if (ms_instance == NULL) {

        ms_instance = new swWxGuiTestProvokedWarningRegistry ();
    }
```

```
    return *ms_instance;
}


void swWxGuiTestProvokedWarningRegistry::Destroy ()
{
    if (ms_instance != NULL) {

        delete ms_instance;
        ms_instance = NULL;
    }
}


void swWxGuiTestProvokedWarningRegistry::RegisterWarning (
        const swWxGuiTestProvokedWarning &warning)
{
    wxASSERT (warnings.end () == warnings.find (&warning));
    warnings.insert (std::make_pair (&warning, false));
}


void swWxGuiTestProvokedWarningRegistry::UnregisterWarning (
        const swWxGuiTestProvokedWarning &warning)
{
    wxASSERT (warnings.end () != warnings.find (&warning));
    warnings.erase (&warning);
}


bool swWxGuiTestProvokedWarningRegistry::IsRegistered (
        const swWxGuiTestProvokedWarning &warning) const
{
    return (warnings.end () != warnings.find (&warning));
}


bool swWxGuiTestProvokedWarningRegistry::WasDetected (
        const swWxGuiTestProvokedWarning &warning) const
{
    return false;
}


void swWxGuiTestProvokedWarningRegistry::SetWarningAsDetected (
        const swWxGuiTestProvokedWarning &warning)
{
    wxASSERT (warnings.end () != warnings.find (&warning));
    ProvokedWarningMap::iterator it;
    it = warnings.find (&warning);
    (*it).second = true;
}

} // End namespace swTst
```

```
#-------------------------------------------------------------------------------
First commit
#-------------------------------------------------------------------------------
..FE.EE

D:\CVS\SimplewareFramework\Cxx\WxGuiTesting\CppTest\
        swwxguitestprovokedwarningregistrytest.cpp(80):Assertion
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testTimeout
assertion failed
- Warning is registered

##Failure Location unknown## : Error
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testTimeout
tearDown() failed
- uncaught exception of unknown type

##Failure Location unknown## : Error
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testDetection
uncaught exception of unknown type

##Failure Location unknown## : Error
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testDetection
tearDown() failed
- uncaught exception of unknown type
```

```
Failures !!!
Run: 3   Failure total: 4   Failures: 1   Errors: 3
<RETURN> to continue


//#############################################################################
swWxGuiTestProvokedWarningRegistryTest.cpp 1.2, 2004/12/20 15:15:13, Check timeout in
        IsRegistered ()
swWxGuiTestProvokedWarningRegistry.h 1.2, 2004/12/20 15:15:13, Check timeout in
        IsRegistered ()
swWxGuiTestProvokedWarningRegistry.cpp 1.2, 2004/12/20 15:15:13, Check timeout in
        IsRegistered ()

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistryTest.cpp,v
// Original Author/Owner: reinhold
//
// Revision: 1.2 Name: State: Exp
//
// Date: 2004/12/2015:15:13 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

//Replace several times:
    CPPUNIT_ASSERT_MESSAGE ("Warning is not registered",
            provWarningRegistry.IsRegistered (warning));
//With:
    CPPUNIT_ASSERT_MESSAGE ("Warning is not registered",
            provWarningRegistry.IsRegisteredAndInTime (warning));

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistry.h,v
// Original Author/Owner: reinhold
//
// Revision: 1.2 Name: State: Exp
//
// Date: 2004/12/2015:15:13 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

//Rename method (and change interface) from:
    virtual bool IsRegistered (const swWxGuiTestProvokedWarning &warning) const;
//To:
    virtual bool IsRegisteredAndInTime (const swWxGuiTestProvokedWarning &warning);

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistry.cpp,v
// Original Author/Owner: reinhold
//
// Revision: 1.2 Name: State: Exp
//
// Date: 2004/12/2015:15:13 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

//Rename method and replace implementation from:
bool swWxGuiTestProvokedWarningRegistry::IsRegistered (
        const swWxGuiTestProvokedWarning &warning) const
{
    return (warnings.end () != warnings.find (&warning));
}
//With:
bool swWxGuiTestProvokedWarningRegistry::IsRegisteredAndInTime (
        const swWxGuiTestProvokedWarning &warning)
{
    ProvokedWarningMap::const_iterator it;
    it = warnings.find (&warning);

    if (it == warnings.end ()) {

        return false;

    } else {

        const swWxGuiTestProvokedWarning *warning = (*it).first;
        wxDateTime now = wxDateTime::Now();
        if (warning->GetTimeout ().IsLaterThan (now)) {

            this->UnregisterWarning (*warning);
            return false;
```

```
        } else {

            return true;
        }
    }
}
```

```
#-------------------------------------------------------------------------------
Check timeout in IsRegistered ()
#-------------------------------------------------------------------------------
.F.F.F

D:\CVS\SimplewareFramework\Cxx\WxGuiTesting\CppTest\
        swwxguitestprovokedwarningregistrytest.cpp(47):Assertion
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testUnRegister
assertion failed
- Warning is not registered

D:\CVS\SimplewareFramework\Cxx\WxGuiTesting\CppTest\
        swwxguitestprovokedwarningregistrytest.cpp(71):Assertion
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testTimeout
assertion failed
- Warning is not registered

D:\CVS\SimplewareFramework\Cxx\WxGuiTesting\CppTest\
        swwxguitestprovokedwarningregistrytest.cpp(96):Assertion
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testDetection
assertion failed
- Warning is not registered

Failures !!!
Run: 3   Failure total: 3   Failures: 3   Errors: 0
<RETURN> to continue
```

```
//////////////////////////////////////////////////////////////////////////////
swWxGuiTestProvokedWarningRegistry.cpp 1.3, 2004/12/20 15:18:45, Correct check timeout in
         IsRegistered ()
// Correct stupid time comparison error from:
        if (warning->GetTimeout ().IsLaterThan (now)) {
// To:
        if (now.IsLaterThan (warning->GetTimeout ())) {
```

```
#-------------------------------------------------------------------------------
Correct check timeout in IsRegistered ()
#-------------------------------------------------------------------------------
...FE

D:\CVS\SimplewareFramework\Cxx\WxGuiTesting\CppTest\
        swwxguitestprovokedwarningregistrytest.cpp(108):
Assertion
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testDetection
assertion failed
- Warning was not detected

##Failure Location unknown## : Error
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testDetection
tearDown() failed
- uncaught exception of unknown type

Failures !!!
Run: 3   Failure total: 2   Failures: 1   Errors: 1
<RETURN> to continue
```

```
//////////////////////////////////////////////////////////////////////////////
swWxGuiTestProvokedWarningRegistry.h 1.3, 2004/12/20 15:25:10, Do not unregister warnings
         in IsRegistered() if timout is exceeded, and thought of having corrected
        WasDetected(), but...
swWxGuiTestProvokedWarningRegistry.cpp 1.4, 2004/12/20 15:25:10, Do not unregister
        warnings in IsRegistered() if timout is exceeded, and thought of having
        corrected WasDetected(), but...

//////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistry.h,v
// Original Author/Owner: reinhold
//
// Revision: 1.3 Name: State: Exp
//
```

```
// Date : 2004/12/2015 : 25 : 10 Author : reinhold
//
////////////////////////////////////////////////////////////////////////////

// Do not remove warnings from registry in check method; allows also making it const;
        Change from:
    /*! ...

        If time out inteval is exceeded, the registered warning is unregistered
        for the sake of simplicity.

        \param warning provoked/expected warning to check for registration
        \return true, if warning is registered
    */
    virtual bool IsRegisteredAndInTime (const swWxGuiTestProvokedWarning &warning);
// To:
    /*! ...

        \param warning provoked/expected warning to check for registration
        \return true, if warning is registered and in time
    */
    virtual bool IsRegisteredAndInTime (const swWxGuiTestProvokedWarning &warning) const;

////////////////////////////////////////////////////////////////////////////
// RCSfile : swWxGuiTestProvokedWarningRegistry.cpp,v
// Original Author/Owner: reinhold
//
// Revision : 1.4 Name : State : Exp
//
// Date : 2004/12/2015 : 25 : 10 Author : reinhold
//
////////////////////////////////////////////////////////////////////////////

// Besides making method const, remove:
            this->UnregisterWarning (*warning);


#----------------------------------------------------------------------------
Do not unregister warnings in IsRegistered() if timout is exceeded, and thought
of having corrected WasDetected(), but...
#----------------------------------------------------------------------------
..E.EE

##Failure Location unknown## : Error
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testTimeout
tearDown() failed
- uncaught exception of unknown type

##Failure Location unknown## : Error
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testDetection
uncaught exception of unknown type

##Failure Location unknown## : Error
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testDetection
tearDown() failed
- uncaught exception of unknown type

Failures !!!
Run: 3   Failure total: 3   Failures: 0   Errors: 3
<RETURN> to continue


//############################################################################
swWxGuiTestProvokedWarningRegistry.cpp 1.5, 2004/12/20 15:41:5, Finally all tests are
        working, it would have been faster to not call UnregisterWarning() in destructor
        , but in the long run...
// Correct destruction of registry from:
swWxGuiTestProvokedWarningRegistry::~swWxGuiTestProvokedWarningRegistry ()
{
    ProvokedWarningMap::iterator it;
    for (it = warnings.begin (); it != warnings.end (); it++) {

        this->UnregisterWarning (*((*it).first));
    }
    warnings.clear ();
}
// To:
swWxGuiTestProvokedWarningRegistry::~swWxGuiTestProvokedWarningRegistry ()
{
    ProvokedWarningMap::iterator it = warnings.begin ();
    while (it != warnings.end ()) {
```

```
        this->UnregisterWarning (*((*it++).first));
    }
    warnings.clear ();
}

#-------------------------------------------------------------------------------
Finally all tests are working, it would have been faster to not call
UnregisterWarning() in destructor, but in the long run...
#-------------------------------------------------------------------------------
...

OK (3)
<RETURN> to continue


//##############################################################################
swWxGuiTestProvokedWarningRegistryTest.h 1.2, 2004/12/20 16:42:10, Started finding of
        registerd warning based on caption and message
swWxGuiTestProvokedWarningRegistryTest.cpp 1.3, 2004/12/20 16:42:9, Started finding of
        registerd warning based on caption and message
swWxGuiTestProvokedWarningRegistry.h 1.4, 2004/12/20 16:42:10, Started finding of
        registerd warning based on caption and message
swWxGuiTestProvokedWarningRegistry.cpp 1.6, 2004/12/20 16:42:10, Started finding of
        registerd warning based on caption and message

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistryTest.h,v
// Original Author/Owner: reinhold
//
// Revision: 1.2 Name: State: Exp
//
// Date: 2004/12/2016:42:10 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

// Added another test:
        CPPUNIT_TEST( testFinding );

    ...

    /*! \fn virtual void testFinding ()
        \brief Test case for finding registered warnings.
    */
    virtual void testFinding ();

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistryTest.cpp,v
// Original Author/Owner: reinhold
//
// Revision: 1.3 Name: State: Exp
//
// Date: 2004/12/2016:42:09 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

// Added another test:
void swWxGuiTestProvokedWarningRegistryTest::testFinding ()
{
    swTst::swWxGuiTestProvokedWarning warning (
            _("Direct␣CAD␣Voxelisation␣Warning"), NULL, 2);
    swTst::swWxGuiTestProvokedWarningRegistry &provWarningRegistry =
            swTst::swWxGuiTestProvokedWarningRegistry::GetInstance ();
    provWarningRegistry.RegisterWarning (warning);

    const swTst::swWxGuiTestProvokedWarning * knownWarning =
            provWarningRegistry.FindRegisteredWarning (
            _("Direct␣CAD␣Voxelisation␣Warning"), "bla");

    CPPUNIT_ASSERT_MESSAGE ("Warning␣was␣not␣found",
            knownWarning != NULL);
    CPPUNIT_ASSERT_MESSAGE ("Warning␣is␣not␣registered",
            provWarningRegistry.IsRegisteredAndInTime (*knownWarning));
    CPPUNIT_ASSERT_MESSAGE ("Warning␣was␣already␣detected",
            !provWarningRegistry.WasDetected (*knownWarning));

    const swTst::swWxGuiTestProvokedWarning * unknownWarning =
            provWarningRegistry.FindRegisteredWarning (
            _("Unknown␣Direct␣CAD␣Voxelisation␣Warning"), "bla");
```

```
    CPPUNIT_ASSERT_MESSAGE ("Warning was found",
            unknownWarning == NULL);
}

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistry.h,v
// Original Author/Owner: reinhold
//
// Revision: 1.4 Name: State: Exp
//
// Date: 2004/12/2016:42:10 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

// Added finder method:
    /*! \fn virtual const swWxGuiTestProvokedWarning * FindRegisteredWarning (const
            wxString &caption, const wxString &message) const
        \brief Find first registerd warning fitting the specified caption and message.

        \param caption caption of provoked/expected warnings to find
        \param message message of provoked/expected warnings to find
        \return corresponding registered warning if found, or NULL if there is
            no appropriate one
    */
    virtual const swWxGuiTestProvokedWarning * FindRegisteredWarning (
            const wxString &caption, const wxString &message) const;

////////////////////////////////////////////////////////////////////////////
// RCSfile: swWxGuiTestProvokedWarningRegistry.cpp,v
// Original Author/Owner: reinhold
//
// Revision: 1.6 Name: State: Exp
//
// Date: 2004/12/2016:42:10 Author: reinhold
//
////////////////////////////////////////////////////////////////////////////

// Added empty implementation for new finder method:
const swWxGuiTestProvokedWarning *
swWxGuiTestProvokedWarningRegistry::FindRegisteredWarning (
        const wxString &caption, const wxString &message) const
{
    return NULL;
}


#----------------------------------------------------------------------------
Started finding of registerd warning based on caption and message
#----------------------------------------------------------------------------
....F

D:\CVS\SimplewareFramework\Cxx\WxGuiTesting\CppTest\
        swwxguitestprovokedwarningregistrytest.cpp(119):
Assertion
Test name: swTst::swWxGuiTestProvokedWarningRegistryTest::testFinding
assertion failed
- Warning was not found

Failures !!!
Run: 4   Failure total: 1   Failures: 1   Errors: 0
<RETURN> to continue


//############################################################################
swWxGuiTestProvokedWarningRegistry.cpp 1.7, 2004/12/20 17:3:58, Finding of registerd
        warning based on caption and message is running as well
// Implemented finder method:
const swWxGuiTestProvokedWarning *
swWxGuiTestProvokedWarningRegistry::FindRegisteredWarning (
        const wxString &caption, const wxString &message) const
{
    bool found = false;
    ProvokedWarningMap::const_iterator it = warnings.begin ();
    while ((!found) && (it != warnings.end ())) {

        const swWxGuiTestProvokedWarning *warning = (*it).first;

        if (warning->GetCaption () == caption) {

            if (warning->GetMessage () != NULL) {
```

```
            } else {
                found = true;
            }
        }
        if (!found) {
            it++;
        }
    }
    if (found) {
        return (*it).first;
    } else {
        return NULL;
    }
}
```

```
#-------------------------------------------------------------------------------
Finding of registerd warning based on caption and message is running as well
#-------------------------------------------------------------------------------
....

OK (4)
<RETURN> to continue
```

Listing A.5: Protocol of applying test-driven development.

A.1.5 swWxGuiTesting Sequence Diagram

Figure A.1 shows a simplified sequence diagram of a symbolic test case (application testing) from test driver start to one rather symbolic event simulation within the test case.

Figure A.1: Sequence diagram: Starting running test cases including event simulation.

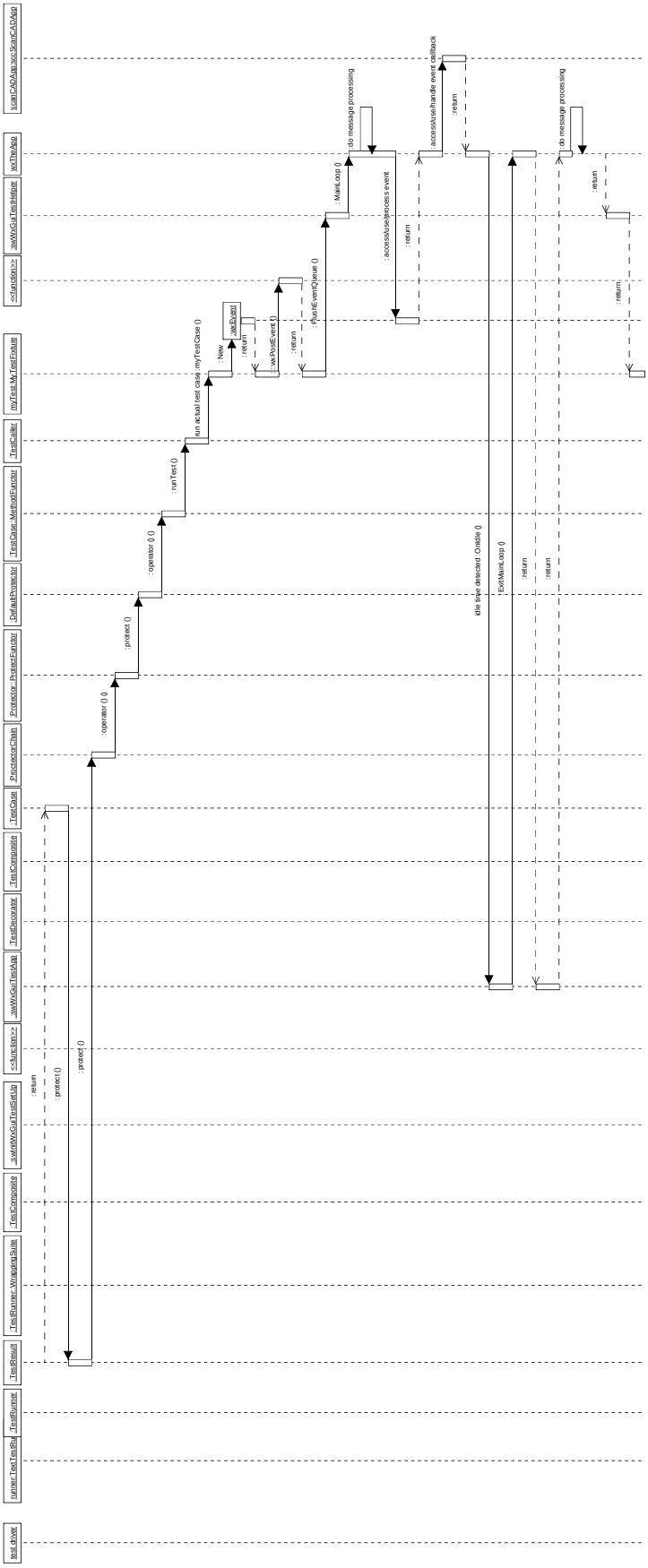


Figure A.1 (continued)

A.1.6 GUI/Application Testing Example

Listing A.6 is an application testing example of importing an STL file into the currently open document.

The test case starts similar to the unit testing example presented in Section A.1.3 by opening a ScanCAD project test file. Main steps and checks carried out subsequently are:

1. Check if loading succeeded.
2. Simulate STL import menu item selection.
3. Check if import dialog has been opened.
4. Set the text control value to point to an STL test file.
5. Confirm settings and close dialog with OK button click.
6. Check if an additional CAD model is now available; and more checks ...

In between the testing is several times temporarily interactive allowing the test case developer visual inspection.

```
void sccCADFileImportGuiTest::setUp () {
  m_cadFileDir = "../../TestData/CAD/";
  // Open a default document:
  wxString scpFilename ("../../TestData/scpv10/quickTest.scp");
  swApp::GetInstance ()->GetDocumentManager ()->CreateDocument (scpFilename,
          wxDOC_SILENT);
}


void sccCADFileImportGuiTest::tearDown ()
{
  // Manipulate document's modify state to close without any user interaction:
  swApp::GetInstance ()->GetDocumentManager ()->GetCurrentDocument ()->Modify (
          false);
  swApp::GetInstance ()->GetDocumentManager ()->CloseDocuments ();
}


void sccCADFileImportGuiTest::testSTLImport ()
{
  const wxString filename = m_cadFileDir + "TessellatedImplant.stl";

  CPPUNIT_ASSERT_MESSAGE ("No document open", swApp::GetInstance ()->
          GetDocumentManager ()->GetDocuments ().GetCount () == 1);
  scc::sccDocument *doc = dynamic_cast< scc::sccDocument *> (swActiveDocument::
          GetInstance ()->GetActiveDocument ());
  CPPUNIT_ASSERT_MESSAGE ("Open document is not active", doc != NULL);
  CPPUNIT_ASSERT_MESSAGE ("Missing CAD model", doc->GetNmbCADModels () == 1);

  wxFrame *appFrame = swFrameFactory::GetInstance ()->GetappFrame ()->GetFrame ();
  wxMenuBar *menuBar = appFrame->GetMenuBar ();
  CPPUNIT_ASSERT_MESSAGE ("Menubar not found", menuBar != NULL);
  int importMenuItemId = menuBar->FindMenuItem (_("File"), _("STL File..."));
  CPPUNIT_ASSERT_MESSAGE ("STL import menu item not found", importMenuItemId !=
          wxNOT_FOUND);
```

```
swTst::swWxGuiTestEventSimulationHelper::SelectMenuItem (importMenuItemId,
        appFrame);
swTst::swWxGuiTestHelper::FlushEventQueue ();

wxWindow *fileSelPanel = wxWindow::FindWindowByName ("sccCADFileSelectorPanel");
CPPUNIT_ASSERT_MESSAGE ("Import dialogs file selection panel window not found",
        fileSelPanel != NULL);
wxTextCtrl *cadFilenameTextCtrl = XRCCTRL (*fileSelPanel, "CADFileTextCtrl",
        wxTextCtrl);
CPPUNIT_ASSERT_MESSAGE ("CAD filename text control not found",
        cadFilenameTextCtrl != NULL);

// Temporary interactive test:
swTst::swWxGuiTestTempInteractive interactive;
interactive.ShowCurrentGui ();

swTst::swWxGuiTestEventSimulationHelper::SetTextCtrlValueWithoutEvent (
        cadFilenameTextCtrl, filename);
swTst::swWxGuiTestHelper::FlushEventQueue ();
interactive.ShowCurrentGui ();

swTst::swWxGuiTestEventSimulationHelper::ClickButton (wxID_OK, fileSelPanel);
swTst::swWxGuiTestHelper::FlushEventQueue ();
interactive.ShowCurrentGui ();

CPPUNIT_ASSERT_MESSAGE ("Invalid number of CAD models - no CAD imported
        successfully", doc->GetNmbCADModels () == 2);
scc::sccCADModelDataset *stlModel = doc->GetCADModel (1);
CPPUNIT_ASSERT_MESSAGE ("Invalid CAD model", stlModel != NULL);
CPPUNIT_ASSERT_MESSAGE ("Wrong filename", stlModel->GetFilename () == filename);

... // Many more checks
```

Listing A.6: Application testing example: importing an STL file.

A.1.7 Captured Event Example

Listing A.7 shows part of the code for capturing text update events.

```
bool swCRTextUpdateEvent::IsPending () const {
  return true;
}


bool swCRTextUpdateEvent::IsIrrelevant () const {
  return m_isIrrelevant;
}


void swCRTextUpdateEvent::Process (swCRCapturedEvent **pendingEvt) {
  wxWindow *wdwEvtObject = wxDynamicCast (m_event->GetEventObject (), wxWindow);
  wxASSERT (wdwEvtObject != NULL);

  // Text updates of spin controls are irrelevant:
  if (wdwEvtObject->GetParent () != NULL) {
    if (wdwEvtObject->GetParent ()->IsKindOf (CLASSINFO(swSpinCtrlDouble))) {
      m_isIrrelevant = true;
      return;
    }
  }

  // If pending event is another text update event on the same (!) control,
  // then destroy it to prevent undesirable code emitting:
  if (*pendingEvt != NULL) {
```

```
    swCRTextUpdateEvent *pendingTextUpdateEvt = dynamic_cast< swCRTextUpdateEvent
            * >(*pendingEvt);
    if (pendingTextUpdateEvt != NULL) {
      if (m_event->GetEventObject () == pendingTextUpdateEvt->GetEvent ()->
            GetEventObject ()) {
        delete *pendingEvt;
        *pendingEvt = NULL;
      }
    }
  }

  swCRWindowHierarchyHandler *hierarchy = swCRWindowHierarchyHandler::GetInstance
          ();
  wxASSERT (hierarchy != NULL);

  if (hierarchy->FindXRCNode (wdwEvtObject) != NULL) {
    m_isXRC = true;
  }
  m_textCtrlName = wdwEvtObject->GetName ();
  m_containerName = hierarchy->FindContainerName (wdwEvtObject);
  wxASSERT (!m_containerName.IsEmpty ());

  wxASSERT (m_event->IsCommandEvent ());
  wxCommandEvent *cmdEvt = wxDynamicCast (m_event, wxCommandEvent);
  wxASSERT (cmdEvt != NULL);
  m_textCtrlValue = cmdEvt->GetString ();
}


void swCRTextUpdateEvent::EmitCpp () {
  // Expected emitting (XRC):
  /*
  wxWindow *evtSimHlpTestPanel = wxWindow::FindWindowByName ("EvtSimHlpTestPanel");
  CPPUNIT_ASSERT_MESSAGE ("Container window for text control 'TextCtrl' not found
          ", evtSimHlpTestPanel != NULL);
  wxWindow *textCtrlWdw = evtSimHlpTestPanel->FindWindow (XRCID("TextCtrl"));
  CPPUNIT_ASSERT_MESSAGE ("Window for text control 'TextCtrl' not found",
          textCtrlWdw != NULL);
  wxTextCtrl *textCtrl = wxDynamicCast (textCtrlWdw, wxTextCtrl);
  CPPUNIT_ASSERT_MESSAGE ("Converting window for text control 'TextCtrl' failed",
          textCtrl != NULL);
  swTst::swWxGuiTestEventSimulationHelper::SetTextCtrlValue (textCtrl, "abc");
  swTst::swWxGuiTestHelper::FlushEventQueue ();
  */
  // Or non-XRC:
  /*
  ...
  wxWindow *textCtrlWdw = evtSimHlpTestPanel->FindWindow ("TextControl");
  ...
  */
  swCRCppEmitter *emitter = swCRCppEmitter::GetInstance ();

  wxString containerVarName = emitter->AddContainerLookupCode (m_containerName,
          wxString::Format ("text control '%s'", m_textCtrlName));

  wxString textCtrlWdwVarName = emitter->MakeVarName (m_textCtrlName, "Wdw");

  wxString str;
  str << "wxWindow *" << textCtrlWdwVarName << " = " << containerVarName << "->
          FindWindow (";
  if (m_isXRC) {
    str << "XRCID(\"" << m_textCtrlName << "\"));";
  } else {
    str << "\"" << m_textCtrlName << "\");";
  }
  emitter->AddCode (str);

  str.Clear ();
```

```
  str << "CPPUNIT_ASSERT_MESSAGE␣(\"Window␣for␣text␣control␣'" << m_textCtrlName
          << "'␣not␣found\",␣" << textCtrlWdwVarName << "␣!=␣NULL);";
  emitter->AddCode (str);

  wxString textCtrlVarName = emitter->MakeVarName (m_textCtrlName);

  str.Clear ();
  str << "wxTextCtrl␣*" << textCtrlVarName << "␣=␣wxDynamicCast␣(" <<
         textCtrlWdwVarName << ",␣wxTextCtrl);";
  emitter->AddCode (str);

  str.Clear ();
  str << "CPPUNIT_ASSERT_MESSAGE␣(\"Converting␣window␣for␣text␣control␣'" <<
         m_textCtrlName << "'␣failed\",␣" << textCtrlVarName << "␣!=␣NULL);";
  emitter->AddCode (str);

  str.Clear ();
  str << "swTst::swWxGuiTestEventSimulationHelper::SetTextCtrlValue␣(" <<
         textCtrlVarName << ",␣\"" << this->GetEscaped (m_textCtrlValue) << "\");"
         ;
  emitter->AddCode (str);

  str.Clear ();
  str << "swTst::swWxGuiTestHelper::FlushEventQueue␣();\n";
  emitter->AddCode (str);
}


wxString CRTextUpdateEvent::GetEscaped (const wxString &str) const {
  wxString escaped;

  for (int i = 0; i < str.Length (); i++) {
    if (str[i] == 92) { // '\'
      escaped.Append ("\\\\");
    } else {
      escaped.Append (str[i]);
    }
  }
  return escaped;
}
```

Listing A.7: Text update event capturing.

A.1.8 VTK Render Window Interaction Compromise

Currently a compromise between full C&R support for VTK render window interactions and "manual" actor-object-oriented coordinate calculation applied on the C&R interaction event recorders appears to leads to maximum reachable maintainability. Listing A.8 shows part of the test case code for selecting the first CAD model in a more object oriented way, where the centre of its bounding box is known to be in front of all other actors in the x view (identified by "rwi2Dx").

```
...
double bounds[6];
doc->GetVisInfo ()->GetProp3DOfDataset (doc->GetCADModel (0))->GetBounds (bounds)
        ;

double centre[3];
centre[0] = (bounds[1] + bounds[0]) / 2.0;
centre[1] = (bounds[3] + bounds[2]) / 2.0;
centre[2] = (bounds[5] + bounds[4]) / 2.0;

vtkCoordinate *coord = vtkCoordinate::New ();
coord->SetCoordinateSystemToWorld ();
coord->SetValue (centre);
int *ivalue;
ivalue = coord->GetComputedDisplayValue (renderers->GetFirstRenderer ());

wxString play;
play << "# StreamVersion 2\n"
    "LeftButtonPressEvent " << ivalue[0] << " " << ivalue[1] << " 0 0 0 0 i
            rwi2Dx\n"
    "LeftButtonReleaseEvent " << ivalue[0] << " " << ivalue[1] << " 0 0 0 0 i
            rwi2Dx\n";
multipleViewRecorder->SetInputString (play);
multipleViewRecorder->Play ();

// Check correct selection:
CPPUNIT_ASSERT_MESSAGE ("Selection failed completely (CAD #1)", dsSel->
        GetSelection ().size () == 1);
CPPUNIT_ASSERT_MESSAGE ("Selection of CAD model dataset failed (CAD #1)", dsSel->
        IsSelected (doc->MakeDatasetKey (doc->GetCADModel (0))));
...
```

Listing A.8: VTK render window interaction capture & replay compromise.

List of Acronyms

AOP ..... Aspect-Oriented Programming

C&R ..... Capture & Replay

CR-TOOL ..... Capture & Replay tool

CCC ..... cross cutting concern

CI ..... Continuous Integration

CUT ..... component under test

DRY ..... Don't Repeat Yourself

GUI ..... Graphical User Interface See UI.

IDE ..... Integrated Development Environment

IUT ..... implementation under test

MSVC ..... Microsoft Visual C++

MVC ..... Model-View-Controller Model-View-Controller is the concept introduced by Smalltalk's inventors of encapsulating some data together with its processing (the model) and isolate it from the manipulation (the controller) and presentation (the view) part that has to be done on a UI.

OO ..... object-oriented

OOP ..... Object-Oriented Programming

OS ..... Operating System

RTTI ..... Runtime Type Information

SCM ..... Software Configuration Management

SUT ..... system/application under test

TDD ..... Test-Driven Development

TFUI ..... TestFirstUserInterfaces

UI ..... User Interface

UML ..... Unified Modelling Language *The* standardised method to design software, i.e. software architecture and behaviour.

UUT ..... unit/class under test

XP ..... eXtreme Programming Probably the most famous agile software development methodology due to being the lightest or most extreme one.

XRC ..... XML-based resource

Bibliography

[AOS] AOSD Steering Committee. Aspect-Oriented Software Development Community & Conference. Internet `http://aosd.net/`. Last visited February 2005.

[App05] Apple Computer, Inc. Apple Human Interface Guidelines. Internet `http://developer.apple.com/documentation/UserExperience/Conceptual/OSXHIGuidelines/`, 2005. Last visited March 2005.

[Bal98] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung.* Object Technology Series. Spektrum Akademischer Verlag GmbH, 1998.

[Bax03] Ira Baxter. Re: test suite for gcc frontend parser? Internet `http://compilers.iecc.com/comparch/article/03-09-023`, September 2003. Last visited February 2005.

[Bec02] Kent Beck. *Test-Driven Development: By Example.* Addison-Wesley Professional, 2002.

[Bee04] Barbar Beenen. JUnit, Teil 2. Komplexe Probleme meistern mit JUnit. Bitte einsteigen! *Java Magazin*, 6(6), 2004.

[Bin01] Robert Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Object Technology Series. Addison-Wesley, third edition, 2001.

[C+] Ward Cunningham et al. Regression Testing. Internet `http://c2.com/cgi/wiki?RegressionTesting`. Last visited January 2005.

[C+04a] Ward Cunningham et al. Gui Testing. Internet `http://c2.com/cgi/wiki?GuiTesting`, 2004. Last visited January 2005.

[C+04b] Ward Cunningham et al. Java Unit. Internet `http://c2.com/cgi/wiki?JavaUnit`, 2004. Last visited January 2005.

[C+04c] Ward Cunningham et al. Test First User Interfaces. Internet `http://c2.com/cgi/wiki?TestFirstUserInterfaces`, 2004. Last visited January 2005.

[C+04d] Ward Cunningham et al. Then Dont Call Main Loop. Internet `http://c2.com/cgi/wiki?ThenDontCallMainLoop`, 2004. Last visited January 2005.

[C+04e] Ward Cunningham et al. Why So Many Cplusplus Test Frameworks. Internet `http://c2.com/cgi/wiki?WhySoManyCplusplusTestFrameworks`, 2004. Last visited January 2005.

[C+05] Ward Cunningham et al. Testing Framework. Internet `http://c2.com/cgi/wiki?TestingFramework`, 2005. Last visited January 2005.

[CAW04] Matt Caswell, Vijay Aravamudhan, and Kevin Wilson. jfcUnit User Documentation. Internet `http://jfcunit.sourceforge.net/`, 2004. Last visited January 2005.

[Cla04] Mike Clark. JUnit FAQ. Internet `http://junit.sourceforge.net/doc/faq/faq.htm\#organize_3`, 2004. Last visited February 2005.

[CM] William E. Caputo and Oren Miller. Continuous Integration with Visual C++ and COM. Internet `http://www.martinfowler.com/articles/ciWithCom.html`. Last visited January 2005.

[Deg05] Degarrah, Inc. Zephyr Automated Test Runner. Internet `http://www.degarrah.com/zephyr.php`, 2005. Last visited May 2005.

[Fau04] Danny Faught. Software Testing FAQ: GUI Test Drivers. Internet `http://www.testingfaqs.org/t-gui.html`, 2004. Last visited January 2005.

[Fea02] Michael Feathers. The Humble Dialog Box. Internet `http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf`, 2002. Last visited January 2005.

[FF] Martin Fowler and Matthew Foemmel. Continuous Integration. Internet `http://www.martinfowler.com/articles/continuousIntegration.html`. Last visited January 2005.

[FHJ+04] Falk Fraikin, Matthias Hamburg, Stefan Jungmayr, Thomas Leonhardt, Andreas Schönknecht, Andreas Spillner, and Mario Winter. Die trügerische Sicherheit des grünen Balkens. *OBJEKTspektrum*, 1, 2004.

[FL] Michael Feathers and Baptiste Lepilleur. CppUnit Cookbook. Internet `http://cppunit.sourceforge.net/doc/1.9.11/cppunit_cookbook.html`. Last visited February 2005.

[Fow00] Martin Fowler. *Refactoring: Wie Sie das Design vorhandener Software verbessern.* Addison-Wesley, 2000.

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. Internet `http://www.martinfowler.com/articles/injection.html`, 2004. Last visited February 2005.

[Ghe03] Andrei Gheorghe. RE: [TFUI] Fw: [XP] User interface. Internet `http://groups.yahoo.com/group/TestFirstUserInterfaces/message/65`, 2003. Last visited February 2005.

[GHJV94] Erich Gamma, R. Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1994.

[HT99] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, first edition, 1999.

[IBM] IBM Rational. Rational Suite TestStudio. Internet `http://www-306.ibm.com/software/awdtools/suite/tstudio/support/`. Last visited January 2005.

[Jef04] Ronald E. Jeffries. Software Downloads. Internet `http://www.xprogramming.com/software.htm`, 2004. Last visited January 2005.

[Kan02] Cem Kaner. Architectures of Test Automation. Internet `http://www.kaner.com/testarch.html`, August 2002. Last visited January 2005.

[Kit] Kitware, Inc. Dart - Tests, Reports, and Dashboards. Internet `http://public.kitware.com/Dart/`. Last visited January 2005.

[Kit05] Kitware, Inc and CMake Community. CMake Testing With CTest. Internet `http://www.cmake.org/Wiki/CMake_Testing_With_CTest`, August 2005. Last visited September 2005.

[Law04] Kevin Lawrence. Re: [TFUI] Lookin for somthing like Jemmy in C#? Anyone knows about this? Internet `http://groups.yahoo.com/group/TestFirstUserInterfaces/message/465`, 2004. Last visited February 2005.

[LD98] Tilo Linz and Matthias Daigl. How to Automate Testing of Graphical User Interfaces. Internet `http://www.imbus.de/forschung/pie24306/gui/aquis-full_paper-1.3.html`, 1998. Last visited January 2005.

[Lep05a] Lepilleur, Baptiste. CppUnit 2 features presentation. Internet `http://cppunit.sourceforge.net/cppunit2/features.html`, 2005. Last visited September 2005.

[Lep05b] Lepilleur, Baptiste. CppUnit2. Internet `http://cppunit.sourceforge.net/cppunit-wiki/CppUnit2`, August 2005. Last visited September 2005.

[Lep05c] Lepilleur, Baptiste. FrontPage - CppUnit Wiki. Internet `http://cppunit.sourceforge.net`, 2005. Last visited January 2005.

[Mer05] Mercury Interactive Corporation. Regression Testing - Mercury WinRunner. Internet `http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/`, 2005. Last visited January 2005.

[MPS00] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. A Planning Based Approach to GUI Testing. Internet `http://www.cs.umd.edu/~atif/papers/SQW2000.pdf`, 2000. Last visited January 2005.

[Mye79] Glenford J. Myers. *Reliable Software Through Composite Design.* John Wiley & Sons, 1979.

[Net] Netbeans Community. Jemmy Module. Internet `http://jemmy.netbeans.org/`. Last visited January 2005.

[Obj04] Object Mentor, Incorporated. JUnit, Testing Resources for Extreme Programming. Internet `http://www.junit.org`, 2004. Last visited January 2005.

[P+] Phlip Plumlee et al. Yahoo! Groups: TestFirstUserInterfaces. Internet `http://groups.yahoo.com/group/testfirstuserinterfaces/`. Last visited January 2005.

[Pas05] Alessandro Pasetti. The XWeaver Project. Internet `http://xweaver.sf.net/`, 2005. Last visited February 2005.

[PKS02] Martin Pol, Tim Koomen, and Andreas Spillner. *Management und Optimierung des Testprozesses: Praktischer Leitfaden für erfolgreiches Software-Testen mit TPI und TMap.* dpunkt.verlag, second edition, 2002.

[PL05] Anthon Pang and Baptiste Lepilleur. wxTestRunner: a CppUnit (C++) Test Runner for wxWidgets. Internet `http://www.softwaredevelopment.ca/wxtestrunner.shtml`, 2005. Last visited February 2005.

[Plu04a] Phlip Plumlee. Re: [TFUI] Lookin for somthing like Jemmy in C#? Anyone knows about this? Internet `http://groups.yahoo.com/group/TestFirstUserInterfaces/message/460`, 2004. Last visited February 2005.

[Plu04b] Phlip Plumlee. Temporary Interactive Tests. Internet `http://groups.yahoo.com/group/TestFirstUserInterfaces/message/567`, 2004. Last visited February 2005.

[pur05] pure-systems GmbH. The Home of AspectC++. Internet `http://www.aspectc.org/`, 2005. Last visited February 2005.

[Qua05] Quality First Software GmbH. qftestJUI - The Java GUI Testtool. Internet `http://www.qfs.de/en/qftestJUI/`, 2005. Last visited May 2005.

[Rob99] John Robbins. Bugslayer, MSJ, April 1999. Internet `http://www.microsoft.com/msj/0499/bugslayer/bugslayer0499.aspx`, April 1999. Last visited January 2005.

[Rob00] John Robbins. Bugslayer: Tester, Take Two-TESTREC.EXE Updates Previous Version of the Tester Utility. Internet `http://msdn.microsoft.com/msdnmag/issues/0600/bugslayer/`, June 2000. Last visited January 2005.

[Rob02] John Robbins. Bugslayer: Tester Utility, Take 3: Adding Mouse Recording and Playback. Internet `http://msdn.microsoft.com/msdnmag/issues/02/03/Bugslayer/default.aspx`, March 2002. Last visited January 2005.

[Sax01a] Marty Saxton. wxTest. Internet `http://wxtest.sourceforge.net`, 2001. Last visited February 2005.

[Sax01b] Marty Saxton. wxTest: A C++ Unit Test Framework. Internet `http://wxtest.sourceforge.net/docs/index.html`, 2001. Last visited February 2005.

[See01] Peter Seebach. The cranky user: Constraining users with modal dialogs. Internet `http://www-106.ibm.com/developerworks/usability/library/us-cranky12.html`, 2001. Last visited February 2005.

[Seg05] Segue Software, Inc. Silktest. Internet `http://www.segue.com/products/functional-regressional-testing/silktest.asp`, 2005. Last visited January 2005.

[SL94] Tim Shea and Jay Lundell. Is there some general rule for when to use modal vs non-modal dialog boxes? Internet `http://edgarmatias.com/faq/S/S-10.html`, 1994. Last visited February 2005.

[The04] The Marathon developers. Marathon - Welcome. Internet `http://marathonman.sourceforge.net/`, 2004. Last visited January 2005.

[Wal04] Timothy Wall. Abbot Java GUI Test Framework. Internet `http://abbot.sourceforge.net/`, 2004. Last visited January 2005.

[Wil04] Anthony Williams. Re: [TFUI] Lookin for somthing like Jemmy in C#? Anyone knows about this? Internet `http://groups.yahoo.com/group/TestFirstUserInterfaces/message/456`, 2004. Last visited February 2005.

[Won04] Franky Wong. Detect and Automatically Respond to User Input Requests from Running Applications. Internet `http://www.desaware.com/tech/detectnewwindow.aspx`, 2004. Last visited February 2005.

[wxW05a] wxWidgets Community. Allocating and deleting wxWidgets objects. Internet `http://www.wxwidgets.org/manuals/2.5.3/wx_allocatingobjects.html`, 2005. Last visited February 2005.

[wxW05b] wxWidgets Community. Application initialization and termination. Internet `http://www.wxwidgets.org/manuals/2.5.3/wx_appinifunctions.html`, 2005. Last visited February 2005.

[wxW05c] wxWidgets Community. wxIdleEvent. Internet `http://www.wxwidgets.org/manuals/2.5.3/wx_wxidleevent.html`, 2005. Last visited February 2005.

[wxW05d] wxWidgets Community. XRC resources format specification. Internet `http://cvs.osafoundation.org/viewcvs.cgi/internal/wxPython-2.5/docs/tech/tn0014.txt?rev=HEAD`, 2005. Last visited May 2005.

[Zei03] Vadim Zeitlin. Re[2]: software simulation of windowing event. Internet `http://lists.wxwidgets.org/archive/wx-users/msg21714.html`, January 2003. Last visited February 2005.