Advanced Unit Testing

How to Scale up a Unit Test Framework

Cyrille Artho*
Nat. Inst. of Informatics
Tokyo, Japan
cartho@nii.ac.jp

Armin Biere
Johannes Kepler University
Linz, Austria
biere@jku.at

ABSTRACT

Unit testing is a scalable and effective way to uncover software faults. In the JNuke project, automated regression tests combined with coverage measurement ensured high code quality throughout the project. By using a custom testing environment, functionality was extended beyond what is commonly available by unit test frameworks. Low-overhead memory leak detection was implemented through wrapping. Automated support for log files made it possible to track the internal state of objects, which is often much more expedient than writing test code. These extensions allowed the easy-to-use unit test framework to scale up to large-scale tests. The techniques can be ported to existing test frameworks.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Measurement, Reliability, Verification

Keywords

Unit testing, system testing, test framework

1. INTRODUCTION

Today, testing is the most widespread way of uncovering faults in software. This is attributed to its applicability to a wide range of projects, its scalability and its effectiveness [19]. In the last few years, unit testing, which targets small, self-contained sections of code, has been widely adopted, mainly thanks to JUnit [15]. JUnit has been developed for Java [10] and has been ported to a large variety of programming languages. It targets testing of individual methods and classes, allowing for a hierarchical structure and fully automated execution of tests. Automation enables *regression testing*, i.e., rerunning tests after each change, which can uncover defects introduced in existing code.

Unfortunately, classical unit testing has several shortcomings, as confirmed by a software developer with several years of experience in industry [11]:

1. Functions that change the state of a complex component are difficult to test individually. Sometimes, it is impossible to tell if a certain internal state is correct.
2. Some tests require a complex context that is difficult to set up.
3. Other tests rely heavily on other modules that are still under development.

The third issue cannot be addressed directly, but JUnit fails to remedy the first two problems as well. Most importantly, it lacks support for log and error log files. Such logging allows tracking the internal state of a component much more easily and is the key to larger-scale *integration testing*, which concerns the interaction between different classes or between entire subsystems. In JUnit, support for file-based test data was omitted on purpose:

> "It generally takes less time in the long run to codify expectations in the form of an automated JUnit test that retains its value over time [13]."

We think that this statement holds for unit tests, but not for integration tests. Codified comparisons are not practical for large, nested data structures. Log files can be analyzed and modified much more quickly. Hence, we found that our extensions allowed us to push the original design of JUnit to cover most kinds of integration tests.

We have used this development environment over four years with seven developers in the JNuke project [2, 5]. The core team included an Assistant Professor and a Ph.D. student, both of which had several years of experience in writing industrial software. Developers worked full-time (as Ph.D. students or Master's students) or part-time (as undergraduate students) on the project, with contributions ranging from two to 18 man-months. The project size amounted to three man-years.

We had established a policy of thorough component testing from the very beginning of the project. Students who participate in such a project often have little practical experience with development of larger systems; strong guidelines helped them to structure their development process and achieve higher-quality results. During development,

*Full address: National Institute of Informatics, Hitotsubashi 2–1–2, Chiyoda-ku, Tokyo 101–8430, Japan

AST '06 May 23, 2006, Shanghai, China.

Java

```
(1) Integer Num;
    int value;
(2) Num = new Integer(8);

(3) value =
      Num.getInteger();
```

JNuke (in C)

```
(1) JNukeObj *Num;
    int value;
(2) Num = JNukeInt_new (mem);
    JNukeInt_set (Num, 8);
(3) value =
      JNukeInt_value (Num);
(4) JNukeObj_delete (Num);
```

Figure 1: JNuke's OO model.

unit tests of former developers also served as documentation. To really take advantage of this, it was indispensible to impose a rigorous code coverage policy, which was accepted by all developers after having gathered initial experience. Despite extra work involved in testing, each component was always delivered on time, and, with a single exception, to specification. Therefore, we will repeat development practices described here in future projects. We think our ideas generalize to other non-interactive systems.

This paper is organized as follows: Section 2 describes JNuke, and how we implemented object orientation in C. Section 3 shows how unit tests are organized, and how advanced features are supported. Section 4 describes our experience with that framework, and Section 5 gives an overview of how test extensions can be applied to other unit test frameworks. Section 6 concludes.

2. JNUKE

JNuke is a framework for verification and model checking of Java programs [2]. It is a novel combination of runtime verification [21], explicit-state model checking [8], and counter-example exploration [22]. Efficiency is crucial in dynamic verification. Therefore JNuke has been written in C, improving performance and memory usage by an order of magnitude compared to competing approaches and tools [5]. Static analysis was added to JNuke at a later stage [2, 3]. At the time of writing, JNuke encompasses about 130,000 lines of code including 1,800 unit tests.

The libraries in JNuke implement some commonly used container classes which are not available by the standard C libraries, such as sets, hash tables, and semi-automatic memory management by reference counting. Iterators with standard semantics are used for data access. This paper only describes testing-related aspects of JNuke. More about JNuke, the technologies used, and their implementation can be read in the corresponding Ph.D. thesis [2].

2.1 Object orientation in C

Thanks to its structured encapsulation of data and actions on data, object-oriented (OO) programming has become the prevalent programming style in the last two decades. Many populaMany popular OO programming languages [10, 16, 17] do not allow the kinds of low-level optimizations available in C or C++ [14, 23]. We decided to create our a custom, simple OO extension for C in order to avoid the complexity of C++ while still enjoying the performance benefits of C. Figure 1 shows a comparison between Java's and JNuke's OO model.

1. In JNuke, any object is generically declared to be of type `JNukeObj *`. There is no possibility to explicitly declare a (statically type safe) object of a specific type. Type data encapsulated in `JNukeObj` includes a pointer to instance data and a pointer to a set of functions. These functions include commonly used operations such as `hash` or `delete`. They were used to implement a low-overhead polymorphism for a few key functions, which were sufficient for most data types. A more fine-grained polymorphism was added later by chaining type data with subtype pointers.

2. Constructors are very similar. In JNuke, operand `new` is written as a function. Each constructor explicitly requires the memory manager as its single argument. Optional arguments in constructors are not possible. Instead, `set` methods have to be used. This choice was taken to simplify serialization and deserialization [6].

3. Statically typed methods are invoked by appending *_methodname* to the class name. The object instance itself (`this` in Java) is always written explicitly as the first argument of the method. Static methods, which do not use instance data, do not need that argument. Polymorphic typing is implemented by dereferencing subtype pointers.

4. Unlike in Java but like in C++, a destructor is required for each class. This method is polymorphic and is wrapped with `JNukeObj_delete`. That function resolves the dynamic type and calls the appropriate destructor.

The callee, which is not shown in the figure, accesses instance data after casting pointer `this` of type `JNukeObj *` to the appropriate subtype. In order to make typecasting more convenient and safer, a macro was implemented that performs an optional type check when resolving the type. Thanks to the flexible preprocessor capabilities, the programmer can use a very compact macro `JNuke_cast (type, reference)` which returns a pointer of the desired subtype pointed to by `reference`.

2.2 String representation of the object state

Like Java library classes, most classes in JNuke also implement their own `toString` function [10]. This function converts the key components of the object state into a human-readable form. Unlike in the Java library, we imposed some strict requirements on the string representation in order to make it machine readable as well. Specifically, any output has to follow a Lisp-like representation, using nested brackets to express the nesting of data structures. We chose this style over XML-like tags for its conciseness, which makes short strings much more readable.

This string representation allowed us to analyze the internal object state at a glance, which was extremely useful for debugging. Moreover, states of objects can be logged easily. Nested data is logged recursively, with brackets indicating the level of nesting. Still, deeply nested data structures are not very readable in this format. To ameliorate this, a pretty printer was created which parses the input string and automatically indents the output according to the nesting level, making even long strings readable [6].

3. UNIT TESTS IN JNUKE

Unit tests in JNuke are organized in several *test suites*. Each package has its own test suite. Test suites are divided

```
int
JNuke_sys_JNukeInt_0 (JNukeTestEnv * env)
{
  /* creation and deletion */
  JNukeObj *Int;
  int res;

  Int = JNukeInt_new (env->mem);
  JNukeInt_set (Int, 1 << 31);
  res = (Int != NULL);
  res = res &&
    (JNukeInt_value (Int) == 1 << 31);
  if (Int != NULL)
    JNukeObj_delete (Int);

  return res;
}
```

Figure 2: A simple test case.

into *test groups.* Normally there is one group for each class. Within a class, several tests can be registered in its group. This hierarchy is similar to the one in JUnit [15].

Each unit test is a function with a name representing its place in the hierarchy of tests: `JNuke_pkg_class_name`. Test cases are registered at the test driver by package name, suite name, and test name. This gives a three-tiered structure. Figure 2 shows the code of a simple unit test which creates a `JNuke_Int` object and deletes it again, checking the class against simple memory leaks. Each test case is a function taking a single `JNukeTestEnv *` argument and returning an integer. Member `env->mem` of struct `JNukeTestEnv` points to the memory manager. The return value is non-zero for a successful test and zero for a failed one.

When running tests from the command line, a directory-like syntax can be used to choose subsets of all unit tests. Within each hierarchy level, a simple pattern-matching operator allows to specify subsets of test more easily. In addition to that, test cases are always classified as "fast" or "slow". By convention, fast test cases do not take more than 0.01 s to run on an unloaded "fast" workstation (such as the ones used in the research labs). This ensures that a large subset of all unit tests can be run quickly even when extra checking tools, such as valgrind [18], are active. Fast tests were run after each compilation, while slow tests were used to prevent regression errors after a new feature was implemented. The hierarchy can also be used to select only the relevant subset of all tests to run after a code change, allowing for change-sensitive testing.

3.1 Testable design

Our development process is bottom-up and test-centered. In particular it always tries to maximize testability even if it implies increased code complexity. Sometimes this required components to be split into separate entities where this may not have been necessary otherwise. Therefore design itself was guided by the question how the intended functionality can be tested.

On a higher level, any configuration or setting was managed by an intermediate layer that allows for local changes. Unit tests could thus override certain configurations for separate testing. Furthermore, as described above, classes had

```
if (res)
  {
    res = 0;
    if (cond)
      {
        /* do something conditionally */
        ...
        /* perhaps more checks */
        res = 1;
      }
  }
```

Figure 3: A construct ensuring maximal coverage.

to support a special string representation in order to facilitate testing. Some classes also included internal consistency check functions for the same purpose. Such extensions only served the purpose of testing.

The fact that code should be designed to be testable also expresses itself at statement level. Testability includes the ability to execute all statements in unit testing, avoiding dead code. Therefore, some of the test code was following certain idioms to achieve this. Typically, a unit test starts by initializing an integer `res` to `1`, which indicates a successful test up to this point. When tests are being run, variable `res` is updated each time a condition is checked. The idiom `res = res && cond` ensures that `res` is set to `0` upon failure. At the same time, this line will always be executed for successful tests, ensuring full statement coverage in that case. If it is not possible to express `cond` in a single line, a construct like the one in Figure 3 should be used. This ensures that there are no statements which are skipped when the test case is successful and makes it possible to achieve 100 % statement coverage for a successful test.

Code using `switch` and `case` statement was similarly modified in order to be usable for coverage measurement. Figure 4 shows an example. If the allowed value range of the `switch` argument is limited, good coding practice checks against illegal values. Out-of-range values should never occur in a correct program. Therefore the program is to be aborted when this happens, because the failure is so severe that the program state is undefined at this point. The left hand side of Figure 4 shows how such code that guards against corrupt states is usually written.

In this example, the `default` clause calling `abort` creates an artefact in the program that only serves failure detection. In a successful run, this statement is never executed. Statement coverage of such code therefore never reaches 100 %. Fortunately, there is a way around this problem. The `default` clause can by used for the last *correct* case instead, replacing the final `case` statement. An assertion is used instead of the `abort` call to ensure that data values are always in the expected range. The right-hand side of Figure 4 shows this new implementation. Note that the code behaves in exactly the same way; error detection is carried out by the `assert` statement. Unlike in the first version, though, executing each correct case leads to full statement coverage, without any artefacts being uncovered.

3.2 Log Files

The success of a test case is determined by its return value. In addition to that, the output of the standard error chan-

```
switch (ternary)
  {
  case 0:
    // first case
    break;
  case 1:
    // ...
    break;
  case 2:
    // last case
    break;
  default:
    // illegal value!
    abort();
  }
```

Normal way of using `switch`/`case`.

```
switch (ternary)
  {
  case 0:
    // first case
    break;
  case 1:
    // ...
    break;
  default:
    assert (ternary == 2);
    // last case
  }
```

Modified code allowing for full statement coverage.

Figure 4: Modifying `switch` statements for coverage.

nel and log files written can also be used to determine the outcome of a unit test. The location of a log files mirrors its place in the hierarchy of the test cases. If a given output file (template) exists, the output of a test will be compared to this file after execution. If the output does not match, or the current test produced no output where an output was expected, the test fails. We call this application of logging *verified logging*.

This logging facility goes beyond functional descriptions commonly used to validate test data. It allows testing of large data structures by writing their content to a log file and then manually inspecting it. If the data is correct, the file is simply renamed and used as a template in the future. This prevents the need for codifying each data element stored, and greatly simplifies the creation of certain complex test cases. It was the key for allowing us to run and evaluate large-scale tests in a unit test framework. Even if this representation of a state became too complex to analyze in detail, it could be used to compare changes in equivalent states between two test runs. Because changes can be analyzed with standard text-based tools such as `diff` and are usually much smaller than the entire state, this state logging allowed for much easier change management.

When using this approach, it is crucial that any such a data dump is performed in a text-based format. While a graphical representation would often be easier to read for a human, the text-based format is a good compromise between human-readability and machine-readability. By using the pretty printer module, the output can be formatted for better readability. The formatted output also allows for an element-by-element comparison using commonly available command line tools.

We used this methodology to analyze complex changes in data structures such bytecode sequences before and after the inlining of subroutines [4]. As instruction indices typically are no longer comparable, filters such as the Unix tools `tr` and `sed` were sometimes used to preprocess the log files. After that, one could pinpoint structural changes easily. Then, it could be decided whether the new output corresponded to a correct change of the algorithm, or whether the change in output was introduced due to an error in development. Without our text-based verified logging facility, we would have had to resort to bytecode disassemblers for viewing data. Therefore verified logging precludes the need for auxiliary tools to handle complex binary data, which are often only kept in memory and never written to a file.

3.3 Memory Management

For a better control of memory management, the standard memory allocation functions have been replaced. The key difference is that the size of allocated blocks is managed by the caller, not by the library. This can save space and make allocations of small blocks more efficient. Because block size is managed by the library user, functions `JNuke_realloc` and `JNuke_free` also expect as an extra argument the size of the currently allocated block at the address the pointer refers to. When compiling JNuke with the standard options, the size of each allocated memory block is stored internally and validated when it is reallocated or freed. When compiling optimized code, that validation is turned off, resulting in a performance improvement over the standard memory allocation library, which maintains that data on its own.

This extra size information is used in each unit test to ensure the absence of memory leaks. When a memory leak is detected, the test is repeated using a memory analysis tool such as valgrind [18] or purify [12], to find the precise location of the allocation that caused the memory leak.

3.4 Coverage measurement

The tight integration of coverage measurement with the GNU C compiler [9] allowed us to automate coverage measurement across several platforms. Statement coverage evaluation detected untested and, hence, possibly dead code. Further testing of such code finds potential faults early. While many kinds of faults cannot be detected using statement coverage alone, we found that statement coverage is achievable with a suitable effort and detects many faults. While other, stricter, measures for coverage exist [19], they are much harder to achieve, not yet supported as well by tools, and not as well understood by developers. Therefore we refrained from using stricter coverage criteria. Instead, we invested time into specifying more properties for test cases. No coverage metrics can detect *missing* code to handle certain unexpected scenarios. Assertions and other consistency checks, on the other hand, can express assumptions and invariants of algorithms, and are just as essential as good test coverage.

Coverage measurement was fully automated: Whenever a class had 100 % statement coverage, it was excluded from the output by a shell script that wrapped the output of the `gcov` tool. This simple post-processing streamlined routine checks after regression testing, because no external coverage viewer has to be invoked when the desired full coverage was reached. Furthermore, per-package coverage measurement is also possible. Testing coverage of a single package does thus not require running the entire test suite.

3.5 Automatic indentation

In order to facilitate editing code, and to improve the quality of the coverage output, automatic indentation was used. After evaluation of several tools, we chose GNU indent because of its wide availability and maturity. The process was eventually fully automated, such that `indent` would be run on all source files prior to each CVS commit command. This decision coincided with the observation that versions 2.2.7

and higher of GNU indent were mature enough such that indentation would always reach a stable point, as opposed to alternating between two possible solutions. While the latter problem does not affect the functionality of the code, it still creates a change which causes a spurious patch to be committed to the code repository after each indentation.

4. EXPERIENCE

How our unit test extensions can be used to facilitate debugging has been described above. This section details our overall experience with the development process, and gives statistics on project progress.

4.1 Development process

Unit tests are the core of the quality assurance effort for JNuke. Work on JNuke was not continuous due to varying student participation and work at NASA during summer 2002 and 2003. Overall, about three man-years of work went into producing JNuke and its unit tests. Due to varying degrees of experience among co-developers, and sometimes rather short participation times, unit tests were essential to keep the project well-documented and well-structured, and to allow for long-term use of crucial modules developed in student projects.

The organizational setting was as follows: An assistant professor and a Ph.D. student were the lead developers, their contributions to the code being roughly 10 % and 40 %, respectively. Another Ph.D. student contributed to certain modules and also supervised student projects. The 40 % contribution by the main developer stretches over four years, amounting to roughly 18 man-months during that time. During this time, and partially in the meantime, undergraduate and Master's students worked on separate modules, contributing between two and six man-months per project. Certain students successively worked on several projects. Student work was usually supervised by weekly or bi-weekly meetings and status reports, regular code inspection, and program analysis with code coverage and memory error detection tools. The key difference to industrial projects was the fact that student projects required a self-contained report to be written. This affected project organization but did not touch actual program development.

Because the projects were graded, supervision by Ph.D. students was limited to important decisions, and except for important quality issues, few implementation problems were directly managed by the supervisors. The students were encouraged to use the unit test process to discover defects on their own. In two cases, our research budget allowed us to hire a student for about two months of part-time work on extra modules. This kind of work-for-hire can be compared to industrial settings.

A requirement specification phase preceeded software design. Due to the open-ended nature of research, many requirements were not absolutely detailed when the initial design was made. Therefore both requirements and design had to be updated as early implementation efforts revealed new problems. Our development model is thus close to the chaos model [20]. This model acknowledges that requirement specification, design, implementation, and testing/maintenance are phases that can never be strictly separated. As a project matures, more effort shifts towards the later stages, but fundamental changes are still needed whenever design decisions have to be revised as a project grows.

This fact motivated a bottom-up, test-centered development process allowing for expedient changes in the design without introducing new errors. A test-driven methodology was used for unit tests, while integration tests were typically added after a given functionality was implemented. This mostly corresponds to the test-first approach from extreme programming [7]. Whenever a task was clear enough in advance such that the outcome could be described in code, test cases to describe the output were created before implementation was started. When this was not possible, such as in larger tests, our test framework was used to execute tests generating a log file. These log files were then inspected and subsequently used for regression testing. Extreme programming uses the term *refactoring* to denote changes in code that have major implications on the design. For instance, a class may be split into several classes, or a fundamental change may require new interactions between objects. Two good examples to describe such occurrences in the JNuke project are multi-threading and garbage collection.

Multi-threaded code may lead to data races when initializing a newly loaded class. A virtual machine (VM) has to use an elaborate protocol to prevent this [10]. Resolving issues related to this became very difficult when rollback and model checking capabilities were added to JNuke. Fortunately they could be fixed after having been revealed by system tests, which were retained as regression tests to ensure the absence of such data races in the future.

Garbage collection had an impact on run-time verification algorithms implemented earlier. Such algorithms may use references to data which is no longer accessible by the running Java program. Because of such references, data may not be garbage collected. Solving this problem required an interaction between the virtual machine and run-time verification event listeners. Previously, the virtual machine had been completely isolated from event listeners.

Large tests used our test harness for system test. Such tests usually include parts of JNuke's own VM. When testing a feature such as dynamic generation of class instances in Java, it is often extremely difficult to write this as a unit test. The reason is that the test setup itself is too complex, requiring initialization of most parts of the VM, loading of several bootstrap classes, and execution of thousands of bytecode instructions to initialize these classes. It is nearly impossible, and certainly not expedient, to "codify" the current state of the VM. Similar reasoning can be applied to the state of the VM after test execution. Instead, our test setup used the VM as a whole: It starts a very small Java program, whose execution produces the desired output if the feature under test is correctly implemented in the VM. As the sample programs can be executed in the standard VM, comparison of JNuke's output to a default output is trivial.

Another adopted practice from extreme programming was the creation of test cases from detected failures. A specialized test case would provide a simpler way of reproducing it. After repairing the defect, a successful test provided a certain confidence in the correctness of the new implementation. Without having a test case to confirm this, one often only relies on intuition to ensure that the program is "fixed".

We also emphasized testing JNuke on several platforms. JNuke runs on the 32-bit and 64-bit variants of Mac OS X, Linux (x86 and Alpha), and Solaris. This revealed programmer errors relating to endianness or assumptions about pointer sizes early.

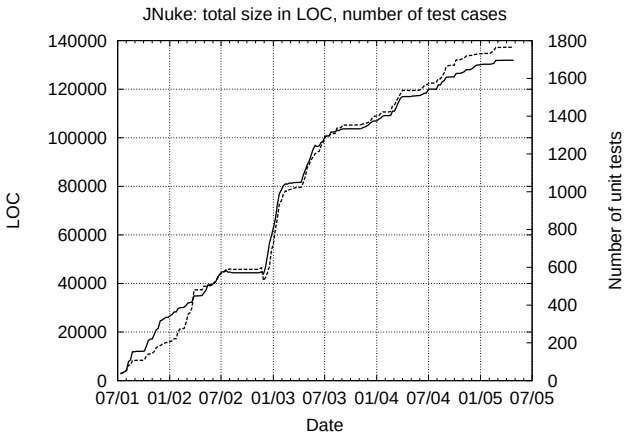

Figure 5: Project size in lines of code (solid) and number of test cases (dashed).

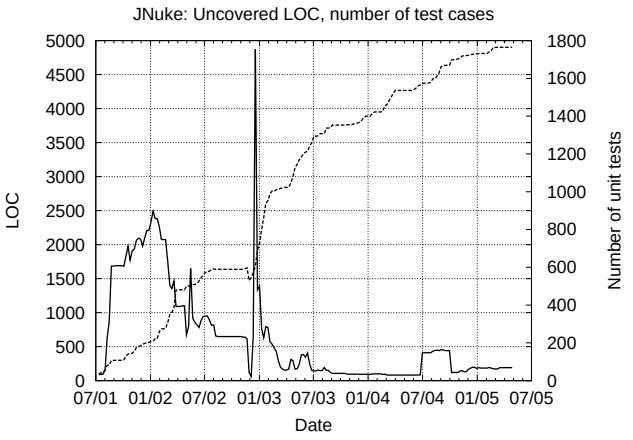

Figure 6: Uncovered lines of code (solid) and number of test cases (dashed).

Because of frequently changing developers, good documentation was vital, not only concerning the high-level design and the tasks solved, but also concerning the concrete implementation. Unit tests are an excellent way of documenting code, providing examples of how to use certain classes or methods.

Although no coverage measurement policy is established by extreme programming, we see it as an essential part of unit testing. In the extreme case, code that has not been tested renders that function unusable once the programmer who wrote it has left the project. Statements that are not covered by tests are often referred to as "dead code". When encountering such dead code, it is not sure whether the code is truly dead code or simply untested. If the code is indeed reachable, then the likelihood of a failure under a future test is very high. Without a prior unit test showing an example of failed code working correctly, it is much more difficult to analyze the failure. Therefore, a strict enforcement of statement coverage is vital for a project in the long run.

4.2 Statistics

As mentioned above, the coverage goal was to achive 100 % statement coverage, in order to find most errors early and to avoid dead code. This goal was almost achieved, with over 99.9 % statement coverage in general and 100 % coverage for those modules that are considered finished.

Figure 5 shows how the project has grown over the nearly four years so far, plotting the overall size of the code against the number of test cases, where about 1800 test cases were used at the end of the project. About 36 % of the code was dedicated to testing. While unit testing was part of the project from its very start, code coverage was not measured initially. The goal of full statement coverage was added later, because locating faults in old code can be very costly, especially when said code was developed by people who no longer participate in the project. Therefore we strongly believe that the effort of preventing such failures pays off in the long term. Figure 6 shows that after a bit more than a year, towards late 2002, a serious attempt was made to achieve (and hold) the goal of full statement coverage. The corresponding reorganization of one module temporarily resulted in a couple of disabled test cases, as can be seen by the short-term increase of uncovered lines. Even though continued development kept introducing new, untested code, care was taken to finish each project with a suite of test cases that achieves full statement coverage.

4.3 Lessons learned

The extensions in our unit test framework allowed us to track internal object states more efficiently, and enabled us to scale the test framework to system-wide tests. This in turn made it possible to impose a strict testing policy. While it took some time for each developer to get familiar with the test framework, the advantages gained eclipsed this minor overhead. Benefits of our strict testing policy include: (1) increased confidence in the implementation, (2) less need for "micromanagement" of projects, (3) facilitated change management, and (4) improved documentation.

The first argument is widely agreed upon, as regression testing prevents old failures from re-occurring. Unit tests also give a formal confirmation that the code fulfills certain properties under given example scenarios.

Test-first development allows developers to work in incremental steps, ensuring that each intermediate goal has been reached. Unit tests validate each implementation step, reducing the need for code reviews and supervision of design changes. Of course these established practices were still carried out frequently at early stages of a project to ensure that quality expectations would be met in the future.

Refactoring allowed us to keep adapting the design to changing requirements. Several new features required major changes, which sometimes had to be performed by developers who had not written the module in question. Thanks to existing unit tests, the changes could be carried out successfully. In this context, unit tests also served as documentation and as a "walkthrough" of the module.

Overall, our experience is that systematic testing on a unit test level as well as integration level allowed us to find many error earlier, led to more maintainable code, while at the same time making other project management practices more effective. We would therefore consider it reckless not to use such a development process in future projects.

5. APPLICATION OF EXTENSIONS TO OTHER TEST FRAMEWORKS

The following key features were added to basic unit testing in the JNuke project:

1. Support for log files (verified logging).
2. Wrapping of memory management.
3. Integrated coverage measurement.

Memory management can be added by using a wrapper for memory management. The test driver has to verify that all memory allocated since the test was started has been freed at the end. Coverage measurement is independent of the test framework used. The idea of minor code style changes to achieve full statement coverage can be carried over to other programming languages as well.

This leaves log file support. Achieving a similar functionality in other test frameworks such as JUnit is not difficult. Each class implements a well-structured representation of its internal state as a text string. This string representation may include only partial information about a state; the key to a successful application of verified logging is that the information must be usable with automated filters while still being human-readable. The implementation of verified logging in the test suite is rather simple. Log files are set up by the test harness before each test is run, and closed thereafter. Comparison of the log output to a template output can be performed after each unit test or after completion of the entire test suite. We believe that many project have already implemented such a convention to some extent, for use in debugging. Therefore such efforts can be re-used for automated testing. When generalizing log files to common test data, there exist similar projects that factor out test input and output data into files. For instance, JXUnit [1] uses XML files to describe test data. However, the usage of files to describe any test data is not widely adapted, because it separates test code from data, which makes unit tests harder to read. We refrained from using files containing serialized objects as test data to avoid this problem.

6. CONCLUSIONS

In the JNuke project, systematic unit testing ensured code quality despite frequent changes of developers. Basic unit testing was extended with rigorous coverage measurement, low-overhead memory leak detection, and verified logging. These techniques allowed us to scale unit tests to larger parts of software, and to utilize the low-overhead unit test infrastructure for system tests as well. Furthermore, coverage evaluation can be fully automated by following simple coding practices. The ideas described in this paper can be ported to other unit test frameworks. Thanks to our success with such unit tests, we will continue to follow the practices described in this document in future projects.

7. ADDITIONAL AUTHORS

Additional authors: Shinichi Honiden (National Insitute of Informatics, Tokyo, Japan, e-mail: `honiden@nii.ac.jp`) and Viktor Schuppan (ETH Zurich, Zürich, Switzerland, e-mail: `vschuppan@acm.org`) and Pascal Eugster (Avaloq Evolution, Zürich, Switzerland, e-mail: `peugster@sysworks.ch`) and Marcel Baur (ETH Zurich, Zürich, Switzerland, e-mail: `marcel.baur@tik.ee.ethz.ch`) and Boris Zweimüller (ETH Zurich, Zürich, Switzerland, e-mail: `zboris@student.ethz.ch`) and Peter Farkas (ETH Zurich, Zürich, Switzerland, e-mail: `pefarkas@student.ethz.ch`).

8. REFERENCES

[1] JXUnit – Java/XML extension of JUnit, 2001. `http://sourceforge.net/projects/jxunit/`.

[2] C. Artho. *Combining Static and Dynamic Analysis to Find Multi-threading Faults Beyond Data Races*. PhD thesis, ETH Zürich, 2005.

[3] C. Artho and A. Biere. Combined static and dynamic analysis. In *Proc. AIOOL 2005*, ENTCS, Paris, France, 2005. Elsevier.

[4] C. Artho and A. Biere. Subroutine inlining and bytecode abstraction to simplify static and dynamic analysis. In *Proc. BYTECODE 2005*, ENTCS, pages 98–115, Edinburgh, Scotland, 2005. Elsevier.

[5] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Proc. CAV 2004*, volume 3114 of *LNCS*, pages 462–465, Boston, USA, 2004. Springer.

[6] M. Baur. Pretty printing for JNuke. Technical report, ETH Zürich, Zürich, Switzerland, 2002.

[7] K. Beck. Test driven development: By example, 2002.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999.

[9] The Gnu Compiler Collection, 2005. `http://gcc.gnu.org/`.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3rd Ed.* Addison-Wesley, 2005.

[11] X. Gu. Personal communication.

[12] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proc. USENIX 1992*, San Francisco, USA, 1992. USENIX Association.

[13] JUnit FAQ, 2005. `http://junit.sourceforge.net/doc/faq/`.

[14] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[15] J. Link and P. Fröhlich. *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann, Inc., 2003.

[16] B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, USA, 1992.

[17] Microsoft Corp. *Microsoft Visual C# .NET Language Reference*. Microsoft Press, Redmond, USA, 2002.

[18] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Proc. RV 2003*, volume 89 of *ENTCS*, pages 22–43, Boulder, USA, 2003. Elsevier.

[19] D. Peled. *Software Reliability Methods*. Springer, 2001.

[20] L. Raccoon. The chaos model and the chaos cycle. *SIGSOFT Softw. Eng. Notes*, 20(1):55–66, 1995.

[21] *1st to 5th Intl. Workshops on Run-time Verification (RV 2001 – RV 2005)*, volume 55(2), 70(4), 89(2), 113, TBD of *ENTCS*. Elsevier, 2001 – 2005.

[22] V. Schuppan, M. Baur, and A. Biere. JVM-independent replay in Java. In *Proc. RV 2004*, volume 113 of *ENTCS*, pages 85–104, Málaga, Spain, 2004. Elsevier.

[23] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1997.