

Software Resurrection: Discovering Programming Pearls by Showing Modernity to Historical Software

Abhishek Dutta

Visual Geometry Group, Department of Engineering Science, University of Oxford
adutta@robots.ox.ac.uk, <https://abhishekdutta.org/sr/>

Abstract—Reading computer program code and documentation written by others is, we are told, one of the best ways to learn the art of writing readable, intelligible and maintainable code and documentation. The software resurrection exercise, introduced in this paper, requires a motivated learner to compile and test a historical release (e.g. 20 years old) version of a well maintained and widely adopted open source software on a modern hardware and software platform. The learner develops fixes for the issues encountered during compilation and testing of the software on a platform that could not have been foreseen at the time of its release. The exercise concludes by writing a critique which provides an opportunity to critically reflect on the experience of maintaining the historical software. An illustrative example of the software resurrection exercise pursued on a version of the SQLite database engine released 20 years ago shows that software engineering principles (or, programming pearls) emerge during the reflective learning cycle of the software resurrection exercise. The concept of software resurrection has the potential to lay foundations for a lifelong willingness to explore and learn from existing code and documentation.

I. INTRODUCTION

This paper introduces the concept of software resurrection as an exercise for discovering software engineering principles for creating maintainable, intelligible and readable computer program code and documentation. This exercise is pursued by a motivated learner who is already familiar with computer programming and wishes to learn the art of writing program code and documentation that requires less maintenance, is easy to understand and enjoyable to read. The exercise begins with the compile stage, as shown in Fig. 1, which requires the learner to compile a historical software release (e.g. released 20 years ago) of a well maintained and widely adopted software on a modern platform. The modern platform, for example, can be a 64 bit multiple core x86 machine running the latest version of operating system (e.g. Linux) and compiler (e.g. GCC-10). The compilation process may require building the dependency libraries. After the compilation succeeds, the next stage requires the learner to test the compiled software using the self contained suite of tests included in the historical release. The learner fixes issues encountered during the compilation and testing stages. The well-defined goal of compiling and testing the historical software on a modern platform keeps the learner motivated. The exercise concludes by writing a critique based on critical reflection of the experiences provided by the compile, test and fix stages of this exercise.

How can, one may wonder, the seemingly pointless activity of compiling, testing and critiquing an old software on a

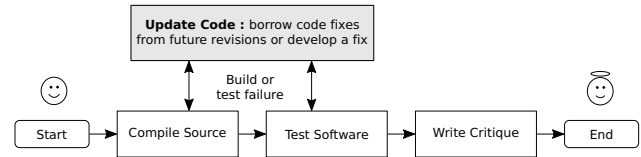


Fig. 1. Software resurrection exercise begins with compilation of a historical software release on a modern platform. After successful compilation, the software’s functionality is verified using automated suite of tests included with the release. Learning opportunities are provided by failure in compilation and testing processes. Learners engage with the program code and documentation to develop a fix for these issues. The exercise concludes by writing a critique of the historical release by reflecting on the experiences gathered during the compilation and testing stages.

modern platform lead to discovery of software engineering principles? The compile and test activities are likely to fail because the developers of the historical software release could not have foreseen the features and constraints of future hardware and software platforms. These failures reveal some important facet of software engineering and provide a thread of investigation to the learner. For example, if the modern compilers have dropped support for a non-standard feature that was widely used and supported 20 years ago then historical software relying on such non-standard feature would not compile on a modern platform, thereby revealing the cost of relying on non-standard features of a compiler. The software resurrection exercise also requires the learner to develop a fix for any issues encountered during the exercise. To fix an issue, the learner must read and understand the program code and documentation contained in the historical software release. To make the exercise more challenging, the learner can decide to not borrow code fixes from future revisions of the software. This provides the learner with a first hand experience of software maintenance. For example, if the program code is well structured and clearly documented, it will simplify the process of fixing the issue pointed by a failing test case. On the other hand, a convoluted code structure, missing documentation, unintelligible identifier names, *etc.*, will puzzle the learner and demand more time and require significant effort to develop a fix. Such joys and frustrations are essential to learn the aspects of program code and documentation that survives the test of time and remains intelligible even after many years. These experiences encourage a motivated learner to adopt best practices in software engineering that improves the readability, intelligibility and maintainability of their software.

The software resurrection exercise also requires the learner to write a critique of the historical software release which provides an opportunity to reflect on the experiences gathered during the compilation and testing stages. Every issue encountered by the learner points to an assumption or a decision made by the developers of the historical software release several years ago. The critique activity encourages the learner to not only identify those assumptions and decisions but also develop an understanding of the circumstances under which those assumptions and decisions were made. For example, a failing test may have been caused by the assumption that 32 bits (or 4 bytes) would always be sufficient to address the memory space of a computer. However, such an assumption would not hold true after the shift of computing hardware to 64-bit systems which would demand 8 bytes of storage for memory addresses. From this failing test, the learner will conclude that a software should have the flexibility to adjust to changing hardware constraints if it expects to remain useful far into the future. The learner should also understand the circumstances that led to an assumption or a decision. For example, to use 8 bytes of storage when 4 bytes of storage was sufficient would not have been frugal as computing memory was, most likely, limited and expensive at that time. Furthermore, the learner could also investigate if it were possible, at that time, to write program code that would have been agnostic to the storage requirement of memory addresses. Such well rounded view of an issue allows the learner to understand the factors that contributed to the failure thereby allowing the learner to develop an impartial view towards a software engineering practice.

Philosophy helps understand why the proposed software resurrection exercise is a valuable teaching and learning tool. The software resurrection exercise helps developers to break free “from the tyranny of the here and the now” [1, p.162] by introducing them to program code and documentation that are remote in time. Such forays into historical software releases enables a learner to view things (*e.g.* software engineering practices) from different perspectives. The learner is no longer captive of their personal viewpoints and becomes capable of surveying a wider horizon of ideas thereby contributing to growth in their wisdom. It is this wisdom that emerges as programming pearls [2] that are commonly shared by experienced programmers who get to know about many things that are remote in time or space by the virtue of their long careers spanning various application domains. Bertrand Russell, a 20th century philosopher, remarked that wisdom can be learned by such excursions into “things that are somewhat remote in time or space” [1].

The concept of software resurrection is similar to the “Learning by doing” [3] methodology which is based on the experiential learning theory [4] of teaching and learning. This perspective will help educators adapt the proposed software resurrection exercise for large scale training (for software developers and researchers) and education (for undergraduate and graduate students) programmes. The compilation, testing and update stages allow learners to actively explore the experience of maintaining a software and the critique stage enables

critical reflection of those experiences. This reflective learning cycle helps with abstract conceptualisation of software engineering practices that contribute to intelligible, maintainable and readable program code and documentation. Such learning experiences are generally not provided by existing software engineering education and training programmes. The learning experience can be enriched further by selecting a historical release that:

- is remote in both time (*e.g.* 20 years old) and space (*i.e.* different from the learner’s domain of expertise),
- has a publicly accessible version controlled history of all code revisions, and
- includes a self contained suite of tests to verify its functionality.

The rest of this paper is organised as follows. Section II surveys other work related to the concept of software resurrection. An illustrative example of the software resurrection exercise pursued on a version of the SQLite database engine that was released 20 years ago is shown in Section III. The compile and test stages are described in Section III-A and III-B respectively. The critique stage is described in Section IV. Experiences from this software resurrection exercise are discussed in Section V. The conclusions from this research are presented in Section VI.

II. RELATED WORK

The software resurrection exercise provides an opportunity to engage with historical program code and documentation. Such engagements with things that are remote in time or space has been recognised by many as a valuable learning experience. Harry R. Lewis [5] has compiled a book containing research papers from Computer Science that have shaped the modern age. The goal of this book is to relieve modern day readers of the “misimpression that the established conventions of the [Computer Science] field were handed down to contemporary culture in finished form” [5].

Reading program code and documentation is an integral part of the software resurrection exercise. The compilation and testing stages requires a learner to read the program code and documentation in order to understand the issues that prevent the software from operating in a modern hardware and software platform. Such active engagement with program code written by others has been recommended by many as a method to improve one’s ability to write maintainable and intelligible software. For example, [6] have designed a course for a professional Master program in Software Engineering that teaches students “how to read the code of an existing, large-scale system to become an effective contributing member of its community”.

Programming pearls (or wisdom) are collected by experienced programmers by virtue of their long programming careers and their engagement with software from a wide variety of application areas. Jon Bentley collected a set of such programming wisdom in a book titled “Programming Pearls” and remarked that “these programming pearls have grown from real problems that have irritated real programmers” [2].

Kernigham and Plauger advised programmers to “Write clearly – don’t be too clever” [7] possibly because they had gone through the challenges of understanding a cleverly optimised code and endured the cost of maintaining such unintelligible code. Donald E. Knuth reflected similar sentiments by warning programmers that “premature optimisation is the root of all evil (or at least most of it) in programming” [8]. The software resurrection exercise provides experiences and context that allows a learner to understand the deeper meaning of these programming pearls.

III. SOFTWARE RESURRECTION OF SQLITE-2002

SQLite is a lightweight and portable database engine that has been actively developed since the year 2000 and has seen wide adoption by users [9]. This section describes the resurrection of SQLite version 2.2.1 released in the year 2002 – henceforth referred as *sqlite-2002* – using a Debian Linux 11.4 operating system running on a laptop purchased in 2019. To conform to the conference publication guidelines, this section contains an abridged version of the exercise; an extended version is available in arXiv [10].

A. Compile Source

The *sqlite-2002* is downloaded and compiled using the standard autoconf based `./configure` and `make` commands. The first build issue is related to a breaking change introduced by the GCC compiler.

```
#error "GCC no longer implements <varargs.h>"
#error "Revise your code to use <stdarg.h>"
```

1) *Compiler Drops Support*: The *sqlite-2002* does not compile in gcc-10.2.1 (2021) and autoconf 2.69 (2012) because the SQL statement parser defined in `tool/lemon.c` uses `varargs.h` header file which was deprecated by the gcc compiler since 4.0 (2005) release. The gcc compiler dropped support for `varargs.h` since April 2004 and switched to supporting `stdarg.h` header file to provide the same functionality. The *sqlite* developers must have adapted their code before the compilers implemented this breaking change. Therefore, version control history of *sqlite* should contain a fix in one of the future revisions. The `vararg` issue was fixed only in *sqlite-2.8.1* release by replacing dependence on `varargs.h` with `stdarg.h`. Unfortunately, a fix for this issue did not appear in a single version control revision (or commit) and the code updates have to be selectively borrowed from the *sqlite-2.8.1* release.

2) *Name Conflict with Standard Library*: After resolving the `varargs.h` issue, the compilation proceeds and reveals the second issue caused by naming conflict with the standard library.

```
error: conflicting types for 'getline'
src/shell.c:50 and /usr/include/stdio.h:616
```

The `getline()` method seems to have been defined in the *sqlite* source (*i.e.* `src/shell.c`) as well as the standard library (*i.e.* `stdio.h`) thereby causing a conflict. If the `getline()` method were a part of the standard library at the time of release, the authors would have renamed their version

of `getline()` before the release to avoid such conflicts. Therefore, the standard library must have been updated after the release of *sqlite-2002*. It is highly likely that one of the code revisions in the version control history of SQLite may contain a fix for this issue as the SQLite software would have adapted to this change in the standard library. A search of the version control system of *sqlite* for the keyword “`getline()`” returns only one result which corresponds to the revision that resolved the name conflict. The conflict resolution involved renaming the method to `local_getline()`. The *sqlite-2002* source compiles successfully after applying the patch generated from the *sqlite* code repository.

B. Run Tests

The *sqlite-2002* is tested using the standard autoconf based `make test` command. An output like “All tests passed” generated by the test command provides assurances that the software behaves in the expected way. However, the test command fails to compile because the Tcl library required to build the tests is missing. The autoconf’s configure script – created in 2002 – is responsible for locating all the dependencies required to compile the tests. This script is unable to recognise the more recent version of Tcl library that is installed using the operating system’s package manager. Therefore, the script that compiles all the tests (*i.e.* `Makefile` which gets generated by the configure script) is manually updated such that the `TCL_FLAGS` and `LIBTCL` variables point to the Tcl library installed by the operating system.

1) *Breaking Changes Introduced by a Dependency*: The build system is able to locate the Tcl library. However, the latest version of Tcl library appears to be incompatible as revealed by the following compilation error.

```
"Tcl_Interp" has no member named "result"
```

The Tcl library seems to have introduced a breaking change because of which the `result` field is not available in the `Tcl_Interp` data structure. The `Tcl_Interp` API documentation describes this breaking change and requires users of this legacy feature to define the `USE_INTERP_RESULT` macro in order to enable access to the `result` field. This issue gets resolved by defining the required macro as advised by the API documentation. The tests compile successfully after an `extern` qualifier is added to the declaration of a variable flagged as undefined by the compiler.

2) *A 32 Bit Software in a 64 Bit System*: Tests compile successfully but they fail to execute on a modern platform due to a `SEGFAULT` error which is caused by programs trying to access a forbidden memory location. The GNU Debugger (gdb) backtrace functionality provides some insight into this issue.

```
Program received signal SIGSEGV
(gdb) backtrace
#0  sqliteBtreeCursor (pBt=0x555e3db0, ...)
    at ../sqlite/src/btree.c:823
#1  btree_cursor (argv=0x55555588a80, ...)
    at ../sqlite/src/test3.c:527
```

The backtrace output suggests that something is amiss with the two pointer variables `pBt` and `argv`; the length

of memory address location stored in them are different. An arduous debugging session reveals that the issue is caused by program code that incorrectly converts the btree pointer address to string representation by wrongly assuming that memory addresses are 32 bits long. This assumption was true in the year 2002 when 32 bits were sufficient to address all the available memory locations. In a modern 64 bit platform, memory addresses are represented by 64 bits (*i.e.* 8 bytes). This issue requires fix in two places: first when a pointer address is converted to string representation and second when the string representation is converted back to a pointer address. The string representations are used by the Tcl script to operate on a test database. To address the first issue, the `%p` format specifier (instead of `%x` which assumes a 32 bit argument) is used to represent the 64 bit pointer address as string. The second issue is addressed by using `strtoul()` function to convert the string representation back to a pointer address as shown below. Such fixes have to be applied at multiple places in `src/test{1,2,3}.c` source files as shown below.

```
static int btree_open(...)
{
    //sprintf(zBuf, "0x%x", (int)pBt);
    sprintf(zBuf, "%p", pBt);
}
static int btree_pager_stats(...)
{
    //if(Tcl_GetInt(interp, argv[1], (int*)&pBt))
    // return TCL_ERROR;
    pBt = strtoul(argv[1], NULL, 16);
    if(!pBt) return TCL_ERROR;
}
```

The `SEGFAULT` error continues to show up during the testing process. Further `gdb` traces reveal that the `src/sqliteInt.c` source code also assumes that pointer variable can be represented by an `int` variable which does not hold true in 64 bit systems. All the tests run successfully to completion after applying the following fix.

```
//# define INTPTR_TYPE int
# define INTPTR_TYPE long
```

IV. CRITIQUE

A. Change is the only constant in a software.

A class of updates to a software that will prevent normal operation of other software tools or services that depends on the software is called a *breaking change*. While a breaking change is undesirable, it is often essential. The Issue III-A1 has revealed that it is important to have flags or markers that caution the users of such breaking changes at the point of usage. The GCC compiler developers have wisely chosen to include a `varargs.h` file in all GCC compiler distributions – since 2004 – which produces an informative error message when the compiler attempts to use the unsupported `varargs.h` header file. The developers of the Tcl library could not provide information about a breaking change at the point of usage. Therefore, to find a fix for Issue III-B1, a maintainer has to dive deep into the TCL library’s documentation.

B. To depend or not to depend is a profound question that the wise can answer.

The SQLite developers chose to rely on a non-standard feature (*i.e.* `varargs.h`) provided by the compilers of their time

(*i.e.* year 2002). The Issue III-A1 revealed that dependence on non-standard features makes the software vulnerable to changes in the ecosystem thereby increasing maintenance costs. Issue III-B1 revealed another facet of software dependencies which often gets overlooked until the dependencies introduce breaking changes. The SQLite software uses the TCL library to implement its test suite. One can understand the benefit of this dependence; it allows the SQLite developers to easily write tests in the Tcl scripting language which is more concise, clear and easier to maintain. The cost of such dependence is often overshadowed by the benefits. All dependencies have a cost and understanding the cost is the first step in taking a wise decision on whether to depend on a third party library or to develop your own functionality.

C. Unitary (or atomic) revisions recorded in a version control system are more useful.

It is easier to understand and reason about a software code or documentation revision that introduces only one conceptual change (*e.g.* a feature, a bug fix, a new test case, *etc.*) in a software. Such revisions can be said to be unitary (or atomic) as they reflect a unit of change in the software. The value of such unitary revisions were realised while fixing Issue III-A2. The `getline()` identifier name conflict with the standard library was fixed as a single revision in the version control history of SQLite. This revision was easy to locate because its revision log contained the `getline` keyword. Such a unitary revision was not available for Issue III-A1 whose fix was more difficult to develop as the fix required manually selecting code updates from one of the future revisions. Therefore, unitary (or atomic) revisions with a revision log containing all the relevant keywords are more useful.

D. Global Identifier Names Should be both Unique and Intel-ligible

The developers of `sqlite-2002` came up with the `getline()` method name well before the method was defined in the standard library through the `<stdio.h>` header file. They fixed this name conflict by renaming the method to `local_getline()` which not only avoided conflict with the `getline()` method in the standard library but also avoided any future conflicts with other software. This shows that uniqueness and intelligibility of identifier names that live in the global namespace can often be achieved with a wisely chosen prefix.

E. Tests define the expected behaviour of a software

The `sqlite-2002` documentation states that this version of `sqlite`, “implements a large subset of SQL92” standard and allows “atomic commit and rollback protect data integrity”. It is possible to prepare some SQL query statements based on the SQL92 standard and prepare some SQL tables to test atomic commits. However, will these be sufficient to confirm that the software is truly behaving in the way it was designed to operate during its release in the year 2002? Furthermore, how can one possibly know what was the desired behaviour

of sqlite-2002 as envisaged by its developers 20 years ago? The sqlite-2002 release includes a set of self contained and automated tests which allow one to quickly and easily verify the functionality of the software on a new platform. These tests not only verify functionality but also act as a concrete specification of the expected behaviour of the software as envisaged by its developers 20 years ago.

F. Who tests the Test?

The SEGFAULT error raised by the original sqlite-2002 test suite demanded an investigation into its architecture. Fig. 2 illustrates the control and data flow for a test case whose purpose is to verify the sqlite engine’s ability to create a new database and represent it using a btree data structure stored in a disk file. The coordination between the test specification code and the test driver layer relies on correct exchange of the test database’s pointer address. An arduous debugging process revealed that this exchange of pointer address was failing because the test driver layer assumed that memory addresses were 32 bits long. Such an assumption broke most of the test code because the driver layer wrongly translated 64 bit pointer addresses (e.g. 0x5555555e3db0) to 32 bit pointer addresses (e.g. 0x555e3db0) by dropping the higher 32 bits portion of the address thereby causing segmentation fault error on access of such malformed memory addresses.

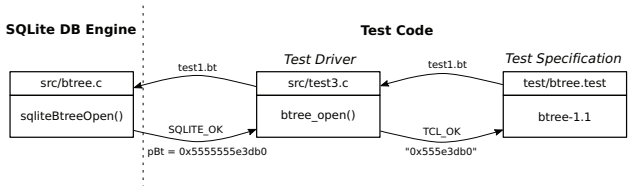


Fig. 2. Test suite architecture of sqlite-2002. The test specifications are defined in the Tcl scripting language. The sqlite core database engine uses the C programming language. A driver layer manages the interactions between the Tcl scripts and the core sqlite database engine.

This issue has revealed that if a test suite has complex logic and involves substantial program code then the test suite merits a testing process for itself to ensure that the test code does not have any flaws. This leads to a recursive testing dependency in which a test suite demands a test for itself in order to assure that the test suite is functioning correctly. Such a requirement can be avoided by a test suite that is so simple that it does not demand a test for itself. The test control and data flow process shown in Fig. 2 must have its merits and therefore was chosen by the sqlite-2002 developers. However, Issue III-B2 has highlighted the need for tests that have minimal setup and simple logic. Only such simple test suites are capable of providing assurances that any failure in the testing process can be solely attributed to the software that is being tested and does not correspond to an issue caused by the test code itself.

V. DISCUSSION

Historical release of the SQLite database engine, released 20 years ago, was compiled and tested in a modern hardware

and software platform to provide an illustrative example of a software resurrection exercise. Some of the issues (e.g. issues III-A1 and III-A2) encountered during this exercise were easy to fix particularly because it was possible to borrow code from future revisions in the version controlled SQLite code repository. These issues required a few hours of independent exploration and learning. Issue III-B2 has been presented in this paper in a condensed form but required many hours (at least 3 weekends) to fix because the explorations often led to a dead end. The software resurrection exercise does require a motivated learner who has the perseverance to systematically investigate an issue, is able to logically reason about possible solutions and can notice the programming pearls (or wisdom) that emerge during critical reflection about the experiences. These requirements expected from the learner can be relaxed by including guidance throughout the reflective learning cycle. For example, learners can be provided with a worksheet that contains a checklist of compilation and testing issues with some tips on fixing those issues. The worksheet can also include a template for the critique if the learners feel a need for some guidance. A limitation of the proposed software resurrection exercise stems from the requirement that the selected historical software should be in a language familiar to the learners. This excludes software and programming languages that do not yet have a long history.

VI. CONCLUSION

This paper has introduced the concept of software resurrection as a teaching and learning tool. The emotions evoked by a well-written program code and documentation is no less pleasurable than great works of literature. An unintelligible program code and documentation is equally useful for learning about the maintenance cost of such creations. The true value of self contained and automated tests is realised when a 20 year old code says “All tests passed” on a modern hardware and software platform that could not have been foreseen by the past. The concept of software resurrection allows one to explore such experiences as well as be aware of many others experiences that can be had by engaging with historical software releases. There is no quantitative experimental data yet to scientifically assess the value and limitations of the concept of software resurrection as a teaching and learning tool; experience reports from educators as well as prospective learners will be required to pursue such an assessment. An illustrative example of the proposed software resurrection exercise provides early indications that it is a valuable tool for learning the art of writing code and documentation that requires less maintenance, is easy to understand and enjoyable to read. The software resurrection exercise has already helped at least one programmer and has the potential to enlighten many more programmers.

ACKNOWLEDGMENT

The author was supported by the EPSRC grant VisualAI (EP/T028572/1) and a research fellowship of the Wolfson College (Oxford University).

REFERENCES

- [1] B. Russell, *Portraits from Memory and other Essays*. George Allen and Unwin Ltd, 1956.
- [2] J. Bentley, *Programming pearls*. Addison-Wesley Professional, 2016.
- [3] G. Gibbs, *Learning by Doing*. Oxford Centre for Staff and Learning Development, Oxford Brookes University, 2013, vol. First Edition.
- [4] D. A. Kolb, *Experiential learning : experience as the source of learning and development*, second edition. ed., Upper Saddle River, New Jersey, 2014.
- [5] H. R. Lewis, *Ideas that Created the Future: Classic Papers of Computer Science*. MIT Press, 2021.
- [6] B. Ryan, A. M. Soria, K. Dreef, and A. van der Hoek, "Reading to write code: An experience report of a reverse engineering and modeling course," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2022, pp. 223–234.
- [7] B. W. Kernighan and P. J. Plauger, *Elements of programming style*. McGraw-Hill, Inc., 1974.
- [8] D. E. Knuth, "Computer programming as an art," *Commun. ACM*, vol. 17, no. 12, p. 667–673, dec 1974.
- [9] K. P. Gaffney, M. Prammer, L. Brasfield, D. R. Hipp, D. Kennedy, and J. M. Patel, "Sqlite: Past, present, and future," *Proc. VLDB Endow.*, vol. 15, no. 12, p. 3535–3547, aug 2022. [Online]. Available: <https://doi.org/10.14778/3554821.3554842>
- [10] A. Dutta, "Software resurrection: Discovering programming pearls by showing modernity to historical software," 2022. [Online]. Available: <https://arxiv.org/abs/2209.05052>