
# Building Better Research Software (Are you sure?): Why Technical Refactoring Does Not Lead To Software Quality

# Building Better Research Software (Are you sure?): Why Technical Refactoring Does Not Lead To Software Quality

Adam Byrne
University of Limerick
Limerick, Ireland
22338004@studentmail.ul.ie

Daniel Kennedy
University of Limerick
Limerick, Ireland
22340017@studentmail.ul.ie

Mark Langtry
University of Limerick
Limerick, Ireland
22340475@studentmail.ul.ie

Art Oliathain
University of Limerick
Limerick, Ireland
22363092@studentmail.ul.ie

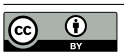
Dominick Stephens
University of Limerick
Limerick, Ireland
22343288@studentmail.ul.ie

Birgit Penzenstadler
Chalmers Tekniska Högskola
Göteborg, Sweden
University of Gothenburg
Göteborg, Sweden
Lappeenranta-Lahti University of Technology
Lappeenranta, Finland
birgitp@chalmers.se

Colin C. Venter
European Organization for Nuclear Research (CERN)
Geneva, Switzerland
c.venters@cern.ch

## Abstract

Software is fundamental to modern research. This paper reports the analysis of source code produced from a Software Carpentry tutorial on Building Better Research Software to understand whether the recommendations and practices advocated led to improvements in the quality of the source code. The analysis was undertaken as part of an educational exercise by fourth year Immersive Software Engineering (ISE) post-graduate students. Their observations suggest (i) the adoption of standard practices in software development such as the use of meaningful variable and function names, removing dead code, the use third-party libraries, separating units of functionality, and using comments and docstrings can lead to code that is easier to read and understand; but also (ii) examination of the code through the lens of core software engineering design principles, i.e. SOLID, highlighted significant violations, which have practical and measurable effects on the quality of the code in relation to key characteristics of maintainability including modularity, modifiability, reusability, and testability. The implications of violating SOLID principles directly undermines maintainability as a core quality that contributes to the overall technical sustainability of all codebases.




*SERS '26, Rio de Janeiro, Brazil*

ACM ISBN 979-8-4007-2404-6/26/04
https://doi.org/10.1145/3786172.3788364

## CCS Concepts

• **Software and its engineering** → **Maintaining software**; **Software evolution**.

## Keywords

Code quality, Maintainability, Modifiability, Modularity, Research Software, Refactoring, Research Software Engineering, Software Design, Software Quality, Software Sustainability

**ACM Reference Format:**
Adam Byrne, Daniel Kennedy, Mark Langtry, Art Oliathain, Dominick Stephens, Birgit Penzenstadler, and Colin C. Venter. 2026. Building Better Research Software (Are you sure?): Why Technical Refactoring Does Not Lead To Software Quality. In *1st International Workshop on Software Engineering and Research Software (SERS '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3786172.3788364

## 1 Introduction

Scientific and engineering research requires a resilient ecosystem of software [7]. A code-first approach to software system development leads to a range of rotten symptoms, including software rigidity, fragility, immobility, and viscosity, which results in high-maintenance and evolution costs, that are the foundation for software decay and death in all software investment [6]. This has resulted in calls for research software to be categorised as a first-class experimental scientific instrument [8]. Software Carpentry [3] was founded in 1998 with the primary aim of improving the productivity of researchers by teaching them core computing skills, such as version control, programming, and task automation [9]. In 2014, Data Carpentry was founded with the mission of building communities teaching universal data literacy. Similarly in 2014, Library

Carpentry was founded with the mission of teaching data skills to people working in library and information related roles. In 2018, Software Carpentry merged with other similar initiatives to form a new project called The Carpentries, under the fiscal sponsorship of Community Initiatives. In 2025, The Carpentries started operations as an independent 501(c)3 non-profit.

This paper reports the insights of an educational exercise as part of a fourth year Immersive Software Engineering (ISE)[1] course on the topic of software architecture for students to understand the implications of a code-first approach to the development of software systems and its impact on architecturally significant requirements such as maintainability. For the purpose of this paper. software architecture is defined as the composition of a set of architectural design decisions (rationale-oriented) [10] that lead to a set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both (structure-oriented) [4]. Similarly, the concept of software sustainability is defined as the capacity of a software system to endure and deliver value over time while minimising negative environmental, economic, and social impacts while remaining adaptable to changing technical, organisational, and societal contexts [5]. Software architecture is the primary lever through which software sustainability is achieved or undermined as architectural decisions determine long-term maintenance effort [6, 11].

## 2 Building Better Research Software

The Carpentries released the Building Better Research Software tutorial [1] as a Beta[2] with the aim to teach practices for producing quality, sustainable and FAIR (Findable, Accessible, Interoperable and Reusable)[3] research software to support open and reproducible research. The lessons are built around an example software project that has not follow "good" software engineering practices and has no documentation. Learners "inherit" code from a pretend team member that has left the team/organisation. The code, written in python, analyses NASA EVA (Extravehicular Activity) data stored in a JSON file. It performs three main tasks: (i) Loads and cleans the EVA dataset, (ii) adds derived information (crew size), and (iii) saves the cleaned dataset as a CSV file and plots a cumulative graph of EVA time.

The tutorial contains eight episodes including *Code Structure*, which explores how using [common code structures] such as... can improve readability, accessibility and reusability of code[4]; see episode for further details. Although not explicitly stated, the course aims to address common maintainability issues including: poor naming conventions, lack of modularisation, i.e. violation of SOLID[5] principles, complex and deeply nested logic, i.e. code smells, duplicated code, i.e. violation of DRY principle, and insufficient comments and documentation. Upon completion of the tutorial, leaners have a [single unit] of python source code[6] that has been [improved] at a technical level incrementally over the course of the lessons by following the [good] software practices the course teaches.

## 3 Immersive Software Engineering (ISE) Assignment

The Immersive Software Engineering [2] program at the University of Limerick is a four year integrated Master of Science degree that radically breaks with traditional models and delivery of software engineering programs. ISE features an intensive student experience with a strong focus on problem solving. The course is taught through the calendar year instead of through semesters, structured with five work placements in which industry partners compete for students. It replaces modular instruction with integrated block teaching. Instruction takes place within a flexible, dedicated space built to carry out these activities with no lecture halls or seminar rooms. Student numbers are kept deliberately small, and instructor numbers are high relative to Irish norms. The core curriculum was designed with industry partners over a one year period and has evolved thereafter.

Thirteen MSc. Immersive Software Engineering students (ISE), three women and ten men, completed the tutorial during two, two-hour sessions. The students were divided into four teams and were tasked with seeing any recommendations through the lens of SOLID; a set of five object-oriented design principles that can be used to design modular, modifiable, scalable, and testable code. While these principles are a fundamental part of object-oriented design best practices they are language agnostic[7]. The students have considerable applied software engineering knowledge and programming expertise having completed four paid residencies as part of their programme, each lasting between three and six months long. The results of each teams analysis were collated into a slide deck under each SOLID principle and presented to the class for discussion to identify similarities and differences. The results are presented in the following section.

## 4 Software Engineering Observations

The code violates the *Single Responsibility Principle*[8]. The code, a [single unit], mixes Command-Line Interface (CLI) handling, orchestration, data I/O, and plotting responsibilities. Print statements are also used for signalling progress, which couples business flow with presentation and logging. The implications of this are: harder to locate and fix bugs because multiple concerns are entangled (Maintainability); unit tests become large and brittle because they must execute unrelated logic (Testability); refactoring risks regressions, since responsibilities are coupled (Reliability). For example, main() manages file I/O, analysis, and plotting. A change to plotting could easily break file-writing.

[1] Immersive Software Engineering: https://software-engineering.ie/

[2] A Beta release is defined as a lesson where the content has been tested by its developers. The beta label is typically applied to a lesson when its developers are confident that it is ready to be used in workshops by other instructors. However, further changes are likely, as feedback is incorporated from additional pilot workshops.

[3] Barker, M. et al (2022). Introducing the FAIR Principles for research software. Scientific Data, 9 (622), DOI: https://doi.org/10.1038/s41597-022-01710-x

[4] https://github.com/carpentries-incubator/bbrs-software-project/tree/05-code-structure/spacewalks

[5] Martin, R. C. "Design principles and design patterns." Object Mentor 1.34 (2000): 597.

[6] Final version: https://github.com/carpentries-incubator/bbrs-software-project/tree/final/spacewalks

[7] Python is an object-oriented language, which allows code to be structured using classes and objects for better organization and reusability.

[8] The Single Responsibility Principle states that "A class should have one, and only one, reason to change."

The code violates the *Open/Closed Principle*[9]. Input/Output locations and formats are hard-coded (defaults to data/eva-data.json, results/... and calls a specific plotting routine). Extending to other formats or plot styles would require code modification rather than extension. The implications of this are: every new feature causes ripple edits across the codebase (Modifiability); frequent modification of core logic increases the possibility of introducing new bugs (Regression Risk); difficult to evolve the system without touching stable components (Scalability). For example, code edits are required inside main() to support CSV input or multiple plot styles.

The code does not violate the *Liskov Substitution Principle*[10] as it is not meaningfully applicable, i.e. no class hierarchy or interfaces. As written, there are no base types to substitute. However, a future risk arises if plotting/IO logic is embedded into concrete functions that later get refactored into classes without clear contracts. The implications of this are: replacement objects break assumptions, leading to subtle runtime bugs (Reliability); code cannot be reused safely across types (Polymorphism); mocks or alternative implementations behave differently from production code (Testing). For example, if a future PlotStrategy subclass saved plots differently (e.g., required network access), old code expecting a local file would fail silently.

The code does not violate the *Interface Segregation Principle*[11] as the codebase does not have any interfaces, but is a likely future pitfall. In its current form, it acts as a "God interface", e.g., a single module handling read, clean, plot, and save. The implications of this are: changes to one feature cascade into all implementers (Coupling); low cohesion causes confusion about what a class is responsible for (Cohesion); mocking large interfaces becomes cumbersome (Testing). For example, a single "EVAService" class that reads, cleans, writes, and plots would force every client to import all dependencies (pandas, matplotlib) even if it only needed reading.

The code violates the *Dependency Inversion Principle*[12]. The (main) method depends on low-level details including concrete file paths, a specific plot function, print/stdout. There is no inversion via constructor injection or factories; dependencies are implicitly global and hard-coded. The implications are: components cannot be reused in other contexts because they depend on environment details (Reusability); unit tests require real files or network calls; mocks cannot be injected (Testability); replacing an implementation (e.g., local to cloud storage) requires rewriting business logic (Flexibility). For example, main() directly constructed file paths and called plt.show(). This hardwired dependency makes it impossible to reuse or unit-test the code without side effects.

## 5 Summary and Conclusions

The paper explored how the practices in a software carpentry tutorial held up against the SOLID design principles as one way of assessing code quality. Overall, the final [improved] code[6] is tightly coupled, with overlapping responsibilities, components cannot be reused without modification, every change touches multiple files/functions, tests need real data and environment, small code edits result in change propagation effects, and it is difficult to safely extend or debug. This analysis strongly suggests that technical refactoring at the code level cannot be exclusively viewed in isolation of conceptual refactoring at an architectural level and does not lead to improved software qualities such as maintainability.

[9]The Open/Closed Principle states that "Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

[10]The Liskov Substitution Principle states that, "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

[11]The Interface Segregation Principle states that "Clients should not be forced to depend upon interfaces they do not use."

[12]The Dependency Inversion Principle states that "High-level modules should not depend on low-level modules. Both should depend on abstractions."

## Acknowledgments

A special thanks to all the participating Immersive Software Engineering students for their contributions and insights: Desiree Charles Dos Santos, John Foley, Dervla Gargan, Conor Glynn, Darragh Grealish, Amy McMahon, Dara Newsome, and Oisin O'Sullivan.

## References

[1] [n. d.]. Building Better Research Software. https://carpentries-incubator.github.io/better-research-software/. [Acc. 16-10-2025].
[2] [n. d.]. Immersive Software Engineering. https://software-engineering.ie/. [Acc. 16-10-2025].
[3] [n. d.]. Software Carpentry. https://software-carpentry.org/. [Accessed 16-10-2025].
[4] Len Bass, Paul Clements, and Rick Kazman. 2021. *Software Architecture in Practice, 4th Edition.*
[5] Christoph Becker et al. 2015. Sustainability Design and Software: The Karlskrona Manifesto. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. 467–476. doi:10.1109/ICSE.2015.179
[6] S. Druskat et al. 2025. Better Architecture, Better Software, Better Research. *Computing in Science & Engineering* 27, 2 (2025), 45–57. doi:10.1109/MCSE.2025.3573887
[7] Z Durdik et al. 2012. Sustainability guidelines for long-living software systems. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 517–526. doi:10.1109/ICSM.2012.6405316
[8] C. Goble. 2014. Better Software, Better Research. *IEEE Internet Computing* 18, 05 (2014), 4–8. doi:10.1109/MIC.2014.88
[9] S. Smith. 2018. Beyond Software Carpentry. In *2018 IEEE/ACM 13th International Workshop on Software Engineering for Science (SE4Science)*. 32–39.
[10] Hans van Vliet and Antony Tang. 2016. Decision making in software architecture. *Journal of Systems and Software* 117 (2016), 638–644.
[11] Colin C. Venter et al. 2018. Software sustainability: Research and practice from a software architecture viewpoint. *Journal of Systems and Software* 138 (2018), 174–188. doi:10.1016/j.jss.2017.12.026