DRAFT

SOFTWARE ENGINEERING STANDARDS FOR EPIDEMIOLOGICAL MODELS

Jack K. Horner
John F. Symons

Abstract

Policy decisions that involve the danger of serious harms require us to deliberate as carefully as is feasible. Often, policy decisions must be informed by models and simulations. Here we argue that to achieve confidence in pandemic policy-making in particular, the epidemiological software we use to inform our policies should be funded, specified, developed, and managed with the same formality and rigor that aircraft-flight-control software systems are. We describe (at a high level) a widely used software engineering standard capable of promoting that rigor. Within that framework, we then assess the most influential modeling and simulation efforts during the current COVID-19 pandemic. We urge modelers and their funding sources to adopt that standard for the development of these critically important software systems. High software engineering standards should be mandated by funding agencies and norms from high-risk engineering contexts should be adopted in scientific contexts with direct and significant public policy implications.

1.0 Introduction

Policy decisions that involve the danger of serious harms require us to deliberate as carefully as is feasible. We rely on expert advice with good reason: there are a wide range of contexts in which our best available sources of evidence and our best guidance for decision-making comes from experts. In such contexts, it is rational to follow our best science in order to increase the likelihood that our decisions promote our values and interests. While science cannot tell us what we ought to value or what our policy goals ought to be, it can help us to achieve our goals and to live up to whatever moral or political principles we have settled upon.

In complex decision-making contexts, policy-makers often rely on simulators developed by teams of scientists (Boschetti et.al 2012). But how do we know which simulations and which models to rely upon? There are many dimensions of this question (see, for example, Symons and Alvarado 2019). In this paper we argue that a necessary condition for justifiable use of simulations in high-stakes decision-making settings for public policy is that they be funded,

specified, designed, implemented, and maintained in accordance with the best available software engineering practices. These practices are as important to software-intensive policy-making as good experimental methods are in non-software-intensive scientific and engineering regimes. Our recommendations will increase the cost of these modeling efforts and will require increased collaboration between scientists and software engineers. The gravity of decisions based on epidemiological modeling efforts warrants the additional resources and effort that we recommend here.

In Section 2, we describe at a high level the consensus software engineering practices framework the software engineering community has determined to help minimize development cost and schedule, while minimizing the potential harms of using that software in high-risk operational venues. Within that framework, we evaluate the conformance of a publicly accessible simulator archive (ICL 2020c) that is closely related to a COVID-19-oriented simulator widely used in pandemic policy-making.

2.0 Software engineering standards in pandemic policy-making

Since late 1960s, the software engineering community has sought to codify consensus software development practices and procedures that have been (empirically) determined to help minimize development cost, schedule, risk (both developmental and operational), and to help ensure that the products of such projects reflect user needs and values (Boehm 1973; Myers 1976; Boehm et al. 2000). These codification efforts have produced a series of software engineering standards.[1] One of the most recent and widely used software engineering standards is ISO 2017. Although there is some variation among these standards, they characterize software projects in terms of lifecycle phases, each with formal review and documentation requirements. These phases are:

1. specification
2. logical design
3. physical design
4. implementation
5. test
6. maintenance

The economic and risk-management rationale for a phased approach to development and management is based on two major premises (Boehm 1981, 38):

I. In order to create a "successful" software product, we must, in effect, execute all of the phases at some stage anyway.
II. Any different ordering of the phases will produce a less successful software product.

[1] Such a standard is not a contract; in the absence of a contract, compliance with a standard is therefore voluntary. A contract, however, can make compliance with a standard mandatory.

Rationale (I) follows directly from the need to answer questions like: "What is the software supposed to do?" (Specification phase), "How do we ensure that everyone who helps to develop part the software understands how the software holds together?", especially if not all personnel know all aspects of the system (Logical, and Physical, design phases),[2] and "How will we determine that the software is doing what is supposed to do" (Test phase).

Rationale (II) derives directly from empirical studies of the costs of fixing errors as a function of the phase in which the error was detected and corrected. These studies show that in a large or highly technical software project, an error is 100 times more expensive to correct in the maintenance phase than in the requirements phase (Boehm 1976; Boehm 1981, 40); in small projects, an typical error is 20 times more expensive to correct in the maintenance phase than in the requirements phase.

Each of phases 1-6 imposes requirements on (equivalently, "allocates requirements to") the processes and products of successor phases. In principle, taken end-to-end, the requirements-allocation chain induces a well-defined mapping from the specification, to the as-built product, its documentation, and processes.

It is sometimes claimed that the code alone in a software system determines what that software is *intended* to do (sometimes called the "intended application semantics" of the code). This view is simply mistaken.[3] Only the equivalent of the combination of the specification, the logical design, the physical design documentation, and various test suites, can state of what the code is supposed to do.

There is no guarantee that using a software development process of the kind described in this section will produce an error-free software system.[4] It is all but certain, however, that if such a framework is not used, the system, with very high probability, will contain errors that could have been avoided if such a framework had been adopted (Boehm 1973; Boehm 1976; Myers 1976; Boehm 1981, 40; Horner and Symons 2019a).

---

[2] On average, five after initial deployment of a software system, only 20% the original developers of the software remain on the project (Boehm et al. 2000, 48). 10 years after initial deployment, on average, none of the original developers remain on the project. On small projects, furthermore, the loss of even a single key team member can force the project to restart. Detailed documentation is the only way to mitigate these risks.

[3] Here are two examples of software that appears to have a specific use, but in fact has a quite different one. In the early 1980s, a large US military data-communications system contained a program, P, that appeared to be data-communications code (and in fact had once been used as such). By 1985, however, P was used solely to stress-test the disk drives in the system. Similarly, in the early 1990s a large European research institute owned what looked like (and in fact once was) a nuclear reactor control code. But by 1995, that code was used solely to test the performance of executables produced by Fortran compilers. More generally, *any* program could be used merely to show that the machine on which it runs will in some sense cycle the program, without regard to anything else that program is "supposed" to do.

[4] See Horner and Symons 2019b for a discussion of whether it is even possible, in all cases of interest, to determine whether we have produced error-free software.

The standards allow tailoring (editing), depending on cost, schedule, and risk to property and life. Software whose failure would result in *in*consequential loss of property, life, or revenue (e.g., software developed solely for personal use) can be developed with little formality according to these standards. In contrast, the standards require that software whose failure could result in large loss of property or life (e.g., aircraft or automobile control) must be developed with extensive formality. (For further information, see: Boehm et al. 2000; Hatton 1995; ISO 2017; Koopman 2014; MISRA 2004, 2008; NASA 2004; Rierson 2013; RTCA 2012; and FDA 2002.)

While epidemiological models generally originate in academic venues, policy makers must rely on modelers for guidance during moments when important decisions need to be made. While informal software development is often tolerated in academic contexts, standards must be higher in the case of epidemiological modeling (Horner and Symons 2019 a,b) that is used in public-health policy-making. The epidemiological simulators used in policy-making are typically used in a way that errors in those simulators could lead to substantial loss of property or life, or to other harms of comparable consequence. Because of this, we argue that they should be developed and managed with the formality required by consensus software engineering standards for aircraft and automobile controls.

2.1 A case study

The Imperial College London (ICL) covid-19 simulator is arguably one of the most influential pieces of scientific software in the history of public health policy-making. Its role in informing the responses of policy makers in the United Kingdom and the United States about the COVID-19 pandemic has been widely reported. The ICL covid-19 simulator continues to play a role in pandemic policy-making at the level of national governments in many countries.

During the period from late-March through late-May 2020, we assessed how well the publicly accessible artifacts of the ICL covid-19 simulator project conform to the consensus software engineering standards framework outlined in Section 2.0. Our assessment was based on informed software engineering judgement, reading those artifacts, building and executing some of the code, and applying various analysis tools (identified below) to the artifacts in that archive. The results of the assessment are organized by the software project phase-structure outlined above.

Scope/limitations of the study

To our knowledge, there is no publicly accessible documentation that officially identifies the baseline for the ICL covid-19 simulator project.[5] The lack of any officially designated, publicly

[5] As of 1 June 2020, an ICL covid-19 project website (ICL 2020d) appears to identify the mapping between certain code archives and various team papers and reports. Our analysis revealed, however, that the code archives identified

accessible project baseline (as described in Sections 2.2 – 2.7, below) is a significant obstacle to the replication or analysis of the ICL group's work.

There is, at present, no legal or institutional requirement for the ICL simulator project to make any software-development artifact of that project accessible to the general public. It is not surprising, therefore, that, even if they exist, many of the artifacts identified in the consensus software engineering standards are not publicly available in the ICL simulator project.

It is increasingly common for scientific software to be made available for the purposes of replication. Different scientific disciplines have adopted their own standards for the transparency of code in recent decades (See for example Freeze 2007, Peng 2011). Generally speaking, the trend in scientific practice has been to encourage "open code for open science" to use Steve Easterbook's slogan (2014).[6]

In our judgement, it is highly likely that ICL 2020c is closer to the actual ICL covid-19 simulator project baseline than any other publicly available artifact; accordingly, we chose ICL 2020c, along with the published articles and reports identified in ICL 2020d, as the baseline for the analysis reported here.

Sections 2.2 – 2.7 describe, at a high level, the major features of each phase of the software engineering process described in the standards mentioned in Section 2.0 and assess how well, within the limitations described above, ICL 2020c conforms to those standards. (For further detail on each of these phases, see ISO 2017.)

2.2 Specification Phase

The principal function of the specification phase of a software project is to generate an agreement (called the *specification*) among stakeholders that states what objectives a software system must achieve. The specification identifies the stakeholders and assigns rights, roles, and responsibilities to them.

The task of identifying stakeholders can involve several, and sometimes contending, considerations. In democratic venues, justifiable decision-making typically requires

---

on this website contained modification date/time stamps that are *later* than the issuance dates of these papers and reports. We further discovered that some of the graphics that appeared in the papers and reports referenced on the website were not directly produced by any of the code in the associated code archives. (It is possible, of course, that some of these graphics were produced by applying software that is not identified in the reports/papers or on the website to the outputs of code that does appear in the archives.) It is therefore not possible to infer from this website, or from the papers/reports linked at this website, the identity of the specific code used produce the results reported in associated papers and reports.

[6] There is no free lunch in open software. Maintaining a publicly accessible archive and responding to inquiries about it has non-zero cost and schedule.

transparency and explainability – even insuring, in some cases, some level of lay understanding.[7] Policy makers cannot be expected to be able to evaluate models and simulations at the level of technical detail, but modelers should be transparent with respect to the degrees of uncertainty involved in their predictions. In complex decision-making problems facing policy makers, modelers must properly represent the extent to which their predictions should be believed. Trusting experts is unavoidable and appropriate in certain domains, especially those with high technical content. It is important to note that expertise in technical, scientific, or engineering domains (such as epidemiology), however, does not imply expertise with respect to societal goals and values.[8] These considerations imply that in some policy-making venues, the general public is a stakeholder and thus can legitimately claim a right to have, in a timely way, access to all policy-related artifacts such as simulator rationale, design, and implementation (a view institutionalized, for example, in UK Government Office for Science 2010):

> 73. SACs [Scientific Advisory Committees] and their secretariats should aim to prepare papers in accessible language. Where issues require technical discussion, consideration should be given to separate, and additional, production of a 'lay summary' to ensure that all matters are accessible to all interested parties regardless of specialist knowledge. (UK Government Office for Science 2011, 18)

Part of developing a specification requires negotiation and resolution of trades among stakeholder values. In the COVID-19 pandemic, for example, optimizing on the social-distancing directives/guidelines collides directly with other activities that all but require person-to-person physical contact. In several stakeholder communities, this trade-off (as of mid-2020) has yet to be resolved. The values shaping the development of the simulation should be explicitly stated for future expectation. Self-critical assessments of the degree to which precautionary or other values enter into the choice of parameters, data sources, etc. would be important contributions to our understanding of the meaning of the predictions derived from the simulation.

There is no publicly accessible specification for ICL 2020c. Ideally, future iterations of these simulators ought to be generated according to publicly negotiated specifications. At the very least, the specifications stipulated by the modelers themselves should be made available to the public. Modelers and their funding organizations might protest that these are time-sensitive projects whose urgency precludes such public deliberation, but as we discuss in Section Three, the trust invested in modelers by the public in these contexts means that they must be able to provide a well-articulated and understandable specification.

---

[7] European Union law establishes a right to explanation in relation to the use of technology in important decisions affecting individual citizens. See for example https://eur-lex.europa.eu/legal-content/EN/TXT/?qid=1465452422595&uri=CELEX:32016R0679 Rectital 71 (accessed June 8 2020). French national law establishes the right to explanation in the 2016 *Loi pour une République numérique*. See also Morely, Cowls, Taddeo, and Floridi 2020.

[8] Robert May (1936-2020) was a remarkable exception. See Krebs and Hassell 2020.

2.3 Logical design

The objective of the logical design phase is to generate an abstract description, called a Logical Design Document, of a system that satisfies the requirements of the specification. (For a discussion of what "satisfaction" means in the software development process, see Horner and Symons 2019b.) This abstract description assumes no particular implementation in hardware, software, or human procedures.

Various languages can be used to express the logical design. In current practice, the Unified Modeling Language (see, for example, Rumbaugh, Jacobson, and Booch 1999) is often used for this purpose.

No software is generated during this phase.

There is no publicly accessible Logical Design document for ICL 2020c.

2.4 Physical design

The objective of the physical design phase is to generate a concrete description, typically called the Physical Design Document, or Detailed Physical Design Document, of how specific machines, software, and human processes, and their interactions, will satisfy the requirements allocated to it from 2.3. The software-specific component of the Physical Design Document is often called the Software Design Document, or SDD. (For a detailed description of an SDD, see US Department of Defense 1988.) In practice, this document typically contains ~10 pages per page of source code (assuming ~50 software statements per page of source code).

No software is generated is generated during this phase.

There is no publicly accessible SDD for ICL 2020c. However, a few items that would be contained in an SDD are included in the inline comments of the source code in ICL 2020c.

2.5 Implementation

This phase implements on actual machines, and in software and human procedures, an operational product that satisfies the requirements allocated to it from the Physical Design.

The implementation phase of a software development project identifies (or can inherit from the system specification), among other things, programming ("coding") standards (sometimes called coding "guidelines"). These standards prescribe programming practices that are, and proscribe

practices that aren't, acceptable (primarily because they do, or don't, help to minimize coding errors. For examples of such standards, see Hatton 1995; Perforce 2013; Google 2020).

As built, the source code in ICL 2020c:

a. Is primarily designed to study the effect of "interventions" (e.g., school closings, social distancing) and population-distribution details on the course of a pandemic. (These results were obtained by "manually" analyzing ICL 2020c and reports and papers associated with that archive.)
b. Appears (based on our analysis inline comments in the source code) to have descended from a multi-thousand-statement simulator written in the C language by one developer in the early 2000s.
c. In its current form, the code is almost entirely implemented in the C language subset of C++. For example, ICL 2020c makes no use of C++ classes or type polymorphism. (We came to this conclusion by reading the ICL 2020c source code, and by analyzing the ICL 2020c source code with the documentation tool *doxygen* (van Heesch 2020) and the static source code analyzer *Understand* (Scientific Tools 2020).)
d. Consists of ~1000 declarative/definitional, and ~10,000 executable, statements, distributed across approximated 30 files. Half of these statements are in a single file that contains the source for the simulator's main routine. (We obtained these metrics by applying the static source code analyzer *Understand* (Scientific Tools 2020) to the source code in ICL 2020c.)
e. ~90% of the program ("compilable") units in ICL 2020c have relatively low McCabe complexity (McCabe 1976). Informally, McCabe complexity is a measure of the number of executable paths in the code; it is also a measure of the intelligibility and the maintainability of the code. Statistically speaking, the frequency of errors in software is an increasing function of McCabe complexity (Basili and Perricone 1984). The remaining 10% of the program units in ICL 2020c, which are deeply coupled to the code's input- and parameter- spaces, have extreme complexity. Unfortunately, this extremely-high-complexity subset contains ~50% of all statements in the code (see (d), above). (The complexity metrics were generated by *Understand* (Scientific Tools 2020).)
f. Requires 40-50 input-variables and parameter-assignments to configure (we obtained this metric by "manual" analysis of the ICL 2020c source code and its input files). This complexity cannot be avoided at some level if the model is to assess the effects of even the intervention regimes that have already been deployed by various countries.[9]
g. Is currently being modified in a way that is generally in accordance with at least some of the best available software engineering practices. (This assessment was based on manually analyzing the source code in ICL 2020c and our experience with software engineering standards and practices.) Based on time-stamps in ICL 2020c, the code has experienced, on average, average annual change traffic (number-of-statements-of-

[9]As a result, ICL 2020c is significantly more difficult to comprehend, correctly use (arguably, only the authors of the code can reliably use it), calibrate, and maintain than simpler epidemiological models such as SEIR (for a description of SEIR, see Vynnycky and White 2010; Nowak and May 2000).

software-changed/total-number-software-statements in the system) of ~5%. This fraction is typical software undergoing relatively minor modifications, not of software being wholly re-engineered (Boehm 1981, 543; Boehm et al. 2000, 28).

2.6 Test

This phase shows that the product generated in the Implementation phase (Section 2.5) satisfies all requirements allocated to the software. Testing is typically performed at various software-build levels.

There is no publicly accessible Test Plan, Test Report, or official Regression Test for ICL 2020c. ICL 2020c contains some test files.

2.7 Maintenance

This phase iterates the phases described in Sections 2.2 – 2.6 after the product is deployed, as needed. Maintenance policies and procedures are documented in a Maintenance Manual.

There is no publicly accessible Maintenance Plan for ICL 2020c.

3.0 Discussion and conclusions

The challenge of building software intensive systems that are suitable for scientific inquiry is non-trivial (See Symons and Horner 2014). As Marco Janssen (2017) has noted, there is increasing pressure for computational scientific enterprises to archive code in public repositories. But archiving code alone insufficient: we must also publicly archive and maintain the major documentation products of the software development process described in Section 2.0: the specification, the logical design document, the physical design document, and various test suites. Making code and its documentation public accessible promotes collective progress, reproducibility, and transparency. Such practices are important in science generally, but they are even more important in those areas of scientific research that play a direct role in public policy decisions.

Justifiable policy-making in epidemiological crises such as the current COVID-19 pandemic involves trades among diverse values. Of course, some aspects relevant to policy-making are addressable, at least in principle, through rigorous software engineering practices (e.g., computing the number of infections, deaths, ICU beds, etc). Other objectives, such as the importance of economic prosperity, education, as well as the relative importance of various moral or political rights and obligations, that often arise in policy-making are another matter. It is imperative that policy makers themselves recognize their responsibility to determine the

relative importance of diverse values rather than claiming to simply "follow the science". Comprehensive policy-making schemes must provide a way to make the tradeoffs among these value-objectives in a way that is fully integrated with engineering standards. Consensus software engineering standards, by design, provide a framework for this integration. Transparent articulation (in the specification) of the purpose of the modeling and simulation project will be important for decision-makers in the evaluation of its advice.

ICL 2020c does not satisfy the standards for safety-critical software identified in Section 2. It is clear, however, that the developers/maintainers of ICL 2020c have recently been modifying that simulator in a way that accommodates at least some of the consensus software engineering standards. We applaud those efforts. In all projects of this kind, we urge teams to adopt methods that support transparency, explainability, and reproducibility within the framework of those standards.

4.0 Acknowledgements

This paper benefitted from discussions with Jorge Soberón, Dick Frank, Dick Stutzke, Tony Pawlicki, and Larry Cox.

JFS was supported in part by NSA Science of Security initiative contract #H98230-18-D-0009.

5.0 References

Basili VR and Perricone BT. (1984). Software errors and complexity: an empirical investigation. *Communications of the ACM* 27, 42-52. https://doi.org/10.1145/69605.2085. Open access. Accessed 2 June 2020.

Boehm BW. (1973). Software and its impact: a quantitative assessment. *Datamation*, 48-59.

Boehm BW. (1976). Software engineering. *IEEE Transactions on Computers*, 1226-1241.

Boehm BW. (1981). *Software Engineering Economics.* Prentice-Hall.

Boehm BW et al. (2000). *Software Cost Estimation with COCOMO II*. Prentice-Hall.

Boschetti F, Fulton E, Bradbury R, and Symons J. (2012). What is a model, why people don't trust them and why they should. In M. R. Raupach (Ed.), *Negotiating our future: Living scenarios for Australia to 2050* (pp. 107–118). Australian Academy of Science.

US Department of Defense. (1988). *Data Item Description DI-MCCR-80012A: Software Design Document.* http://continuum.org/~brentb/2167a-did-sdd.html. Accessed 28 May 2020.

Easterbrook, S. (2014). Open code for open science. *Nature Geoscience*, 7: 779-781.

Freese J. (2007). Replication standards for quantitative social science: Why not sociology? *Sociological Methods & Research*, *36*(2), 153-172.

Hatton L. (1995). *Safer C: Developing Software for High-Integrity and Safety-Critical Systems.* McGraw-Hill.

Horner J and Symons, J. (2019a). Understanding Error Rates in Software Engineering: Conceptual, Empirical, and Experimental Approaches. *Philosophy & Technology*, *32*(2), 363-378.

Horner J and Symons J. (2019b). Why there is no general solution to the problem of software verification. *Foundations of Science*. https://doi.org/10.1007/s10699-019-09611-w.

Imperial College London (ICL). (2020a). COVID-19 model, v1.0. https://github.com/ImperialCollegeLondon/covid19model/releases/tag/v1.0. Accessed 20 April 2020.

Imperial College London (ICL). (2020b). Report 9: Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand. https://www.imperial.ac.uk/mrc-global-infectious-disease-analysis/covid-19/report-9-impact-of-npis-on-covid-19/. Accessed 28 May 2020.

Imperial College London (ICL). (2020c). https://github.com/mrc-ide/covid-sim/blob/master/src/. Accessed 10 May 2020.

Imperial College London (ICL). (2020d). COVID-19 scientific resources. https://www.imperial.ac.uk/mrc-global-infectious-disease-analysis/covid-19/covid-19-scientific-resources/ . Accessed 3 June 2020.

Google, Inc. (2020). Google C++ Style Guide. https://google.github.io/styleguide/cppguide.html. Accessed 2 June 2020.

ISO/IEC/IEEE. (2017). *ISO/IEC/IEEE 12207:2017. Systems and software engineering – Software life cycle processes.* https://www.iso.org/standard/63712.html. Accessed 26 May 2020.

Janssen MA. (2017). The practice of archiving model code of agent-based models. *Journal of Artificial Societies and Social Simulation* 20(1).

Knuth DE, Floyd RW. (1971). Notes on avoiding 'GO TO' statements. *Information Processing Letters* 1, 23-31. Reprinted in *Writings of the Revolution: Selected Readings on Software Engineering.* E. Yourdon (ed). Yourdon Press. pp. 153-162.

Koopman P. (2014). *A Case Study of Toyota Unintended Acceleration and Software Safety.* Briefing slides. https://users.ece.cmu.edu/~koopman/pubs/koopman14_toyota_ua_slides.pdf. Accessed 26 May 2020.

Krebs JR and Hassell M. (2020). Obituary: Robert May. *Nature* 581, 261.

McCabe T. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering* (4): 308–320. doi:10.1109/tse.1976.233837. Accessed 27 May 2020.

MISRA. (2004). MISRA C:2004. https://www.misra.org.uk/misra-c/Activities/MISRAC/tabid/160/Default.aspx. Accessed 26 May 2020.

MISRA. (2008). *MISRA C++ Guidelines for the use of the C++ language in critical systems.* https://www.misra.org.uk/Activities/MISRAC/tabid/171/Default.aspx. Accessed 26 May 2020.

Morely J, Cowls J, Taddeo M, and Floridi L. (2020). Ethical guidelines for COVID-19 tracing apps. *Nature* 582, 29-31.

Myers GJ. (1976) *Software Reliability.* John Wiley.

NASA. (2004). *NASA Software Safety Guidebook. NASA.* https://standards.nasa.gov/standard/nasa/nasa-gb-871913. Accessed 26 May 2020.

Nowak MA and May RM. (2000). *Virus Dynamics: Mathematical Principles of Immunology and Virology*. Oxford.

Peng RD. (2011). Reproducible research in computational science. *Science* 334, 1226-1227.

Perforce, Inc. (2013). High Integrity C++ Coding Standard. V 4.0. https://www.perforce.com/resources/qac/high-integrity-cpp-coding-standard. Accessed 2 June 2020.

Rierson L. (2013). *Developing Safety-Critical Software: A Practical Guide for Aviation Software DO-178C Compliance.* CRC Press.

RTCA, Inc. (2012). *DO-178C, Software Considerations in Airborne Systems and Equipment Certification.* RTCA.

Rumbaugh J, Jacobson I and Booch G. (1999). *The Unified Modeling Language Reference Manual*. Addison-Wesley.

Scientific Tools, Inc. (2020). *Understand*. https://scitools.com/. Accessed 26 May 2020.

Symons J and Alvarado R. (2019). Epistemic Entitlements and the Practice of Computer Simulation. *Minds and Machines.* https://doi.org/10.1007/s11023-018-9487-0.

Symons J and Horner J. (2014) Software intensive science. *Philosophy & Technology*, *27*(3), 461-477.

UK Government Office for Science. (2010). Principles of scientific advice for government. https://www.gov.uk/government/publications/scientific-advice-to-government-principles/principles-of-scientific-advice-to-government. Accessed 27 May 2020.

UK Government Office for Science. (2011). Code of Practice for Scientific Advisory Committees. https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/278498/11-1382-code-of-practice-scientific-advisory-committees.pdf. Accessed 27 May 2020.

US FDA. (2002). *General Principles of Software Validation: Guidance for Industry and FDA Staff*. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/general-principles-software-validation. Accessed 26 May 2020.

van Heesch D. (2020). *Doxygen*. http://doxygen.nl/. Accessed 26 May 2020.

Vynnycky E and White RG (eds). (2010). *An Introduction to Infectious Disease Modelling*. Oxford.