Received: 15 March 2021 | Accepted: 23 August 2021

DOI: 10.1111/2041-210X.13716

Methods in Ecology and Evolution
BRITISH ECOLOGICAL SOCIETY

PERSPECTIVE

Dealing with software complexity in individual-based models

Daniel Vedder[1] | Markus Ankenbrand[2] | Juliano Sarmento Cabral[1]

[1]Center for Computational and Theoretical Biology, Ecosystem Modeling Group, University of Würzburg, Wurzburg, Germany

[2]Center for Computational and Theoretical Biology, University of Würzburg, Wurzburg, Germany

Correspondence
Daniel Vedder
Email: daniel.vedder@stud-mail.uni-wuerzburg.de

Funding information
Bayerisches Staatsministerium für Wissenschaft, Forschung und Kunst; European Social Fund, Grant/Award Number: ESF-ZDEX

Handling Editor: Giovanni Strona

Abstract

1. Individual-based models are doubly complex: as well as representing complex ecological systems, the software that implements them is complex in itself. Both forms of complexity must be managed to create reliable models. However, the ecological modelling literature to date has focussed almost exclusively on the biological complexity. Here, we discuss methods for containing software complexity.
2. Strategies for containing complexity include avoiding, subdividing, documenting and reviewing it. Computer science has long-established techniques for all of these strategies. We present some of these techniques and set them in the context of IBM development, giving examples from published models.
3. Techniques for avoiding software complexity are following best practices for coding style, choosing suitable programming languages and file formats and setting up an automated workflow. Complex software systems can be made more tractable by encapsulating individual subsystems. Good documentation needs to take into account the perspectives of scientists, users and developers. Code reviews are an effective way to check for errors, and can be used together with manual or automated unit and integration tests.
4. Ecological modellers can learn from computer scientists how to deal with complex software systems. Many techniques are readily available, but must be disseminated among modellers. There is a need for further work to adapt software development techniques to the requirements of academic research groups and individual-based modelling.

KEYWORDS

ecological modelling, individual-based models, model complexity, research software engineering, software complexity, software development

1 | INTRODUCTION

After a slow start in the 1980s, individual-based models (IBMs, also known as agent-based models or mechanistic simulation models) have found growing acceptance as a tool of ecological research (DeAngelis & Grimm, 2014; Grimm & Railsback, 2005). Over the past decade, they have been continuously expanded to simulate increasingly complex macroecological (e.g. Cabral & Kreft, 2012; Harfoot et al., 2014), eco-evolutionary (e.g. Leidinger et al., 2021; Schiffers et al., 2014) and macroevolutionary dynamics (e.g. Cabral et al., 2019; Hagen et al., 2021; Rangel et al., 2018), among others.

Although including this complexity is important to increase the predictive power of ecological models (Evans et al., 2013; Stillman et al., 2015), it can also make the models hard to reason about and analyse (Manson et al., 2020). Indeed, it has been argued that increasing complexity is both a necessary development and one of the greatest challenges for the field (Cabral et al., 2017). Following the adage that IBMs 'should be constructed as simple as possible, and as complicated as necessary' (Sun et al., 2016), significant work has been done to develop methodological and analytical techniques for dealing with this biological complexity. These techniques include pattern-oriented modelling (Grimm & Railsback, 2011), 'evaludation' (Augusiak et al., 2014) and cascaded design of simulation experiments (Lorscheid & Meyer, 2016).

However, in its quest to cope with the scientific consequences of model complexity, the ecological modelling community has neglected some of the technical aspects of modelling. IBMs simulate complex ecological systems using computer code (software) that is itself a complex system—in effect, they display a double complexity. Thus, to generate reliable results, we must not only think about the biological complexity of our models but also about their technical complexity (Grimm & Railsback, 2005; Ropella et al., 2002). This is all the more important because unreliable software can have catastrophic consequences for entire research fields (e.g. Eklund et al., 2016; Hatton, 1997).

However, this is an engineering challenge, not primarily a scientific one, and as such many ecologists may feel uncomfortable tackling it (Cooper & Hsing, 2017; Nowogrodzki, 2019). The problem is not restricted to ecology: a growing realisation of the importance of software to science has led to the emergence of the field of 'Research Software Engineering' (Cohen et al., 2021) and several other groups working to improve the software development skills of scientists. In line with these developments, we authors argue that as ecological modellers, we ought to pay more careful attention to the software running our models.

Previous publications have established best practice in regard to the workflow of model development (Grimm et al., 2014; Stodden & Miguez, 2014), suggested methods for scaling large simulations (Parry, 2009), or given general guidelines for scientific programming (Balaban et al., 2021; Wilson et al., 2014, 2017). Therefore, in this commentary, we will specifically look at the question of software complexity in IBMs, with the aim of increasing model reliability. In doing so, we will draw on some of the classical software engineering literature, as well as our own experiences and observations in the development of open-source applications and IBMs (e.g. Ankenbrand et al., 2018; Leidinger et al., 2021; Petter et al., 2021). The individual points will be illustrated using references to published IBMs with publicly available source code (Table 2). We intend this overview to be particularly helpful for graduate students and early career researchers who already have some modelling experience, but would profit from a firmer grounding in the 'tools of the trade' of software development.

Right from its inception, the field of computer science has had to deal with the issue of software complexity (Dijkstra, 1972). As programs became larger and 'programming' turned into 'software engineering', numerous techniques were developed for containing this complexity. These techniques have been applied to language design, coding style, software architecture and development workflow (Brooks, 1986). In fact, this topic is considered to be of such great importance that one well-known textbook states that 'reducing complexity is arguably the most important key to being an effective programmer' (McConnell, 2004, p. 839).

Essentially, there are four strategies that can be used to contain both biological and technical complexities (Table 1):

1. Avoid unnecessary complexity.
2. Subdivide unavoidable complexity.
3. Document existing complexity.
4. Regularly review complexity.

Multiple development techniques target each of these strategies. In the following section, we will briefly present a collection of these techniques and give examples for how they have been applied to ecological IBMs. Subsequently, we will discuss steps that could or should be taken to better integrate good software development practices in the field of individual-based modelling.

TABLE 1 Overview of strategies and techniques for dealing with biological and software complexity in individual-based models. See the main text for explanations and references

Strategies	Techniques	
	Biological complexity	**Software complexity**
Avoid	Pattern-oriented modelling	Clean code
		Language & file formats
		Automated integration
Subdivide	Cascaded design of simulation experiments	Encapsulation domain-specific languages
Document	ODD	README & manual
	TRACE	Source code comments
		Logging
Review	Evaludation	Code reviews
		Unit & integration tests

TABLE 2 Examples of open-source individual-based models and other ecological software that are referred to in the text.

Software	Purpose	Reference	Language	Illustrates
DEBplant	Mechanistic species distribution model	Schouten et al. (2020) https://github.com/rafaqz/DEBplant	Julia	Encapsulation, testing
FENNEC	Trait data collation tool	Ankenbrand et al. (2018) https://github.com/molbiodiv/fennec	PHP, Javascript	Documentation, automated testing
GeMM	Plant community model	Leidinger et al. (2021) https://github.com/CCTB-Ecomods/gemm	Julia	File formats, encapsulation, documentation, integration
Gen3sis	Macroevolutionary speciation model	Hagen et al. (2021) https://github.com/project-gen3sis/R-package	R, C++	Documentation, language interoperability
MadingleyR	Multitrophic ecosystem model	Hoeks et al. (2021) https://github.com/MadingleyR/MadingleyR	R, C++	Encapsulation, language interoperability
RangeShifter	Population range dynamics model	Bocedi et al. (2014) https://github.com/RangeShifter/	C++	Encapsulation, logging
XL/GroIMP	Plant growth model	Hemmerling et al. (2008) https://sourceforge.net/projects/groimp/	Java	Domain-specific languages

2 | PRACTICAL RECOMMENDATIONS

2.1 | Writing code

To avoid unnecessary complexity (Table 1), readability must be the first concern when actually writing model code. In the words of Abelson et al. (1996): 'Programs must be written for people to read, and only incidentally for machines to execute' (p. xvii). Although in some special cases other needs may override this rule (e.g. in performance-critical sections), these must remain justified exceptions. Readability is a core aspect of understandability, which makes programs easier to learn, reason about and analyse. Software that is easy to understand is less prone to mistakes, and therefore more reliable (McConnell, 2004).

An observational study of 78 professional software developers showed that only 5% of their time was spent directly editing the code, whereas 58% was spent understanding the code base and a further 24% navigating it (Xia et al., 2018). Importantly, the developers cited comparatively trivial root causes for this high understandability cost. These included insufficient comments, meaningless variable names, overlong functions and classes, inconsistent coding styles and exaggerated inheritance hierarchies in object-oriented programs.

This highlights the essential nature of 'clean code' (cf. Martin, 2009). Basic practices can make a significant difference, such as dividing code into appropriately sized functions and files, or keeping layouts simple by avoiding long lines or deep nesting. Further guidelines may be found in Wilson et al. (2017), in the style guides published for many programming languages, or (in great depth) in McConnell (2004).

2.2 | Choosing a language

The choice of programming language is often both deeply subjective and contingent upon external circumstances such as available collaborators and libraries. Nonetheless, we believe that it is important to choose the language for a new project carefully, as the decision will influence the project over its entire lifetime. Therefore, based on our personal experiences, we will hazard to give an opinion on a few popular programming languages below, as well as a recommendation for our preferred language.

When choosing a language for a model, one does not want to introduce unnecessary complexity with an overly complicated language (Table 1), yet one must consider the computational requirements of large models. Thus, the ideal language for complex IBMs ought to be one that is easy to program in, while offering a good runtime performance. This, however, casts questions on some common modelling languages. Specifically, C++ is generally perceived as fast but very complicated; it is powerful, but error-prone particularly with regard to its manual memory management. R is great for data analysis, and enjoys the advantage of being the only language that many ecologists are taught in their university courses. However, for individual-based models, it is often both slow and linguistically so complicated that large R programs are very hard to write and even harder to read. Python, on the other hand, can be a good choice, as it is exceptionally easy to learn, write and read, although getting it to run fast requires the use of more advanced numeric libraries. NetLogo also can be a good choice, as it is simple to learn and tailored to the needs of individual-based modellers; despite its former reputation for slowness, improvements over the past years now make it a feasible platform even for large models (Railsback et al., 2017).

After having used all of these languages at various times, our working group has settled on Julia (Bezanson et al., 2017) as our current 'ideal candidate'. Julia is a comparatively new language that is explicitly designed for the requirements of computational scientists. Its syntax is similar to Python and easy to pick up, it supports multiple programming paradigms (including object-oriented and functional programming), and has an excellent performance even without specialised libraries (Lubin & Dunning, 2015). Its rapidly growing popularity among computational scientists has greatly increased the availability of libraries and help resources. Individual-based models can be written directly in the language, or make use of the agent-based modelling framework `Agents.jl` (Datseris et al., 2021). There are also mature libraries available for visualisation, such as `Plots.jl` (Breloff, 2021) or the aforementioned `Agents.jl`. Overall, our experiences with the language have been almost entirely positive (e.g. Leidinger et al., 2021; Sieger & Hovestadt, 2020; Vedder et al., 2020).

Obviously, shifting to a new language represents a significant time investment, both in terms of learning and teaching as well as porting any existing code. Also, the size of the 'language ecosystem' is often crucial: how many libraries are available, how easy is it to find help, do project collaborators know the language too? Indeed, such community constraints often make modellers hesitant to pick up a new language, particularly as most ecologists are only taught R in their university courses. Nonetheless, it should be pointed out that learning a new language is a one-time investment that pays long dividends and may well save time and effort in the long run. In short, we believe that the Julia ecosystem is now large and stable enough, and the long-term benefits great enough, to recommend the increased adoption of Julia for IBMs. For two examples of models written in Julia, see the DEBplant and GeMM models (Table 2; Leidinger et al., 2021; Schouten et al., 2020).

At the same time, we should point out that while the model itself might be best written in Julia, auxiliary software can and should utilise other languages. Programming languages are tools, each with their own strengths and weaknesses. Accordingly, one should choose the best tool for the job at hand. For example, in the GeMM model, we use Python and Bash scripts to set up our simulation experiments, Julia for the actual model code, and R to visualise and analyse the output data (see Section 2.8). In some instances, it may also be worthwhile to use language interoperability libraries such as `Rcpp` to benefit from the strengths of multiple languages. This can be tricky, but has been done for instance in the Gen3sis and MadingleyR models (Table 2; Hagen et al., 2021; Hoeks et al., 2021).

2.3 | Working with file formats

File formats are conventions for how to structure files that are read or manipulated by computer programs. They may be specified by official committees (e.g. PDF, HTML), or be defined ad-hoc for individual programs. IBMs often interact with multiple input and output file types, such as GeoTIFF for map inputs or CSV for configuration files. As different file types have different advantages and disadvantages, it is worth thinking about which formats to use and how to create new ones.

Fundamentally, a format may use binary (e.g. PDF) or text-based (e.g. HTML) data encoding. In the programming tradition that grew up around the Unix operating systems (such as Linux or Mac OS), a premium is placed on using textual file formats wherever possible (Raymond, 2003). Although these may require more storage space than binary formats, they have a number of practical advantages that are directly relevant to the development of IBMs. First, they are human-readable, and can easily be opened with any text editor. This makes debugging a lot simpler, as one is not dependant on special software to analyse model output. Second, text files can be read, analysed and processed by a whole host of other programs and programming languages. This means one can mix and match at will, always using the best tool for the job, instead of being locked into one particular software (see Section 2.2). And third, compared to direct binary 'memory dumps', the level of indirection introduced by exporting to text-based formats automatically encourages encapsulation (see Section 2.4) and helps to keep a model backwards-compatible as it grows. Thus, text-based file formats help to avoid complexity on multiple levels (Table 1). Because of this, we decided to use text-based input and output formats for the GeMM model, which allows us for instance to easily script the creation of configuration files for an experiment, or to output FASTA data for analysis with standard bioinformatical tools (Table 2; Leidinger et al., 2021).

In some cases, programs begin to rely so heavily on their custom-made file formats that these develop into what are known as domain-specific languages, or DSL. This may happen when the program requires extensive and intricate configuration, or the developers use the file format to rapidly add new content (such as player items in a computer game, or species in an IBM). This can be very effective, as using a simple language designed specifically for the problem at hand is more efficient than using a complex general-purpose language (Raymond, 2003). It is also a great way to build abstraction layers and subdivide complexity (see Section 2.4; Table 1), as a DSL represents a clearly defined interface between different sections of the software (Abelson et al., 1996).

Two examples of open-source projects that have used this technique with great success are the LLVM compiler toolchain and the 'Battle for Wesnoth' strategy game (Brown & Wilson, 2011). It has also found application in the XL/GroIMP plant modelling software (Table 2; Hemmerling et al., 2008), which has been the basis for numerous studies (e.g. Chi et al., 2016; Evers & Bastiaans, 2016; Petter et al., 2021). Of course, DSLs are not easy to design well and no easier to implement. Thus, this technique should be avoided for simple models or by inexperienced developers. However, for large models with experienced programmers on the team, they can be an extraordinarily powerful method for creating highly versatile IBMs.

2.4 | Encapsulation

This technique is most important during the software design stage, when constructing the code architecture. Here, the key principle is 'divide and conquer', that is, subdividing a complex system into simpler subsystems (Table 1). This should be done in such a way that each subsystem can be thought about individually, and the operation of the whole does not depend on the implementation details of its constituent parts.

Two complementary approaches can be used for this. The first, 'abstraction', refers to a hierarchical layering of a program. For example, a graphics program may have a fundamental layer that draws individual pixels onto the screen, an intermediate layer for drawing basic shapes and a top layer for rendering entire scenes. Importantly, each layer only needs to know about the one just below it. In the context of IBMs, 'super-individuals' are a common abstraction level (Grimm & Railsback, 2005), used, for example, by the MadingleyR model (Table 2; Hoeks et al., 2021). Similarly, the RangeShifter model (Bocedi et al., 2014) groups individuals into populations, populations into patch subcommunities and patch subcommunities into the landscape-wide community.

The second approach is 'modularity', and refers to a side-by-side encapsulation of program parts. For example, a desktop application may have a submodule responsible for the graphical user interface, and a submodule for processing input. The two modules communicate via pre-defined function calls, but know almost nothing about each other's internal working. Likewise, the GeMM model has submodules devoted to file input and output, which provide utility functions to the rest of the model while hiding the implementation details of the different file formats involved (Leidinger et al., 2021). As another example, the RangeShifter model separates organisms from their environment: there is a `Landscape` object that holds the environment's grid cells, and a `Community` object that contains all individual organisms (Bocedi et al., 2014). DEBplant, in turn, splits its functionality into three independent packages to model dynamic energy budgets, microclimates and photosynthesis (Schouten et al., 2020).

Reducing complexity by encapsulation brings several great advantages. It aids understandability, which, as argued above, also makes software more reliable. Furthermore, encapsulated designs are easier to extend in future, because the strict structure reduces the number of locations where changes need to be inserted. For example, all ecological processes in the GeMM model are contained within individual functions that are passed the `world` object as a function argument. Thus, adding a new process is as simple as writing a new function to modify this object, and inserting it in the desired sequence in the scheduling submodule. For more detailed discussions of encapsulation, see Abelson et al. (1996), Raymond (2003), and McConnell (2004).

2.5 | Documenting models

To maintain understandability of any non-trivial model, adequate documentation is critical (Table 1). As with complexity in general, this must again be thought of in two dimensions. First, the model's formulation must be documented, that is, how it represents the underlying biological reality. Which entities and processes does it simulate, what parameters does it use and what data does it record? Such questions ought to be answered in an accompanying ODD document (Grimm et al., 2020). Further details on the scientific process behind the modelling study, such as problem formulation, data evaluation and model analysis, are collated in the TRACE document (Grimm et al., 2014).

At the same time, one must document the technical details of the model implementation (Cooper & Hsing, 2017; Lee, 2018). This must countenance both the needs of users and the needs of developers of the software. Standard practice in the open-source movement is to provide a README file in the top-level project directory, giving a brief overview over what the software does and how to use it. For large models, a more comprehensive user manual is necessary (such as provided by RangeShifter, Table 2; Bocedi et al., 2014), especially if the model is to be published in a public repository like CRAN. To help developers, there should be an additional document that gives an introduction to the software architecture (Raymond, 2003). In any case, all source files and all functions should have header comments briefly stating their purpose. Many languages also have libraries available to automatically generate cross-referenced HTML or PDF documentation from these header comments. For examples of such automatically generated documentation, see the FENNEC tool and the Gen3sis model (Table 2; Ankenbrand et al., 2018; Hagen et al., 2021).

One non-obvious, yet important, documentation method is a good logging system. Many open-source programs have the ability to save runtime status messages to a log file for later inspection. This can be a valuable help in tracking down bugs but also helps new developers observe the program in action, effectively providing a dynamic documentation. It is useful to make the amount of detail saved in the log file configurable, this is known as the 'verbosity level'. Common verbosity levels include 'quiet' (only report errors), 'normal' (errors and key function calls) and 'debug' (high-detail reporting for use during development). Both the RangeShifter and the GeMM model make use of such logging facilities (Table 2; Bocedi et al., 2014; Leidinger et al., 2021).

Lastly, using version control systems together with hosting platforms like Github can help to document the software development process itself, through the use of commits, branches, issues and the like (Perez-Riverol et al., 2016).

2.6 | Code reviews

Even well-designed and documented software systems cannot be expected to be free of defects and errors. Therefore, reviewing and testing source code regularly is critical (Table 1). Submitting software to internal reviews has been shown to be one of the single most effective means of increasing code quality. Numerous studies have shown that on average, such reviews can be expected to find ~60% of bugs (Shull et al., 2002). This is even more effective than automated testing, and can save significant debugging time down the line. Also, reviews have the added advantage of rapidly disseminating know-how and best practice within teams (see the discussion in McConnell, 2004).

Reviews may be done very formally, with moderated meetings to inspect a section of code using standardised checklists (Aurum et al., 2002). They may also be more informal, using digital tools to quickly review new contributions to a software (Rigby & Bird, 2013). Either way, they require regular time investments, and ideally need to be embedded in a team's development routine. However, the benefits are so great that this investment should pay for itself. Indeed, code reviews have already been recommended for use in IBM development (Ropella et al., 2002), and are provided for in the TRACE protocol as one method of implementation verification (Grimm et al., 2014). To give a practical illustration of how to conduct code reviews, we provide an inspection checklist that we developed for use in our institute (Vedder, 2019).

External code reviews can also be an important supplement to the normal peer-review process for publications. For example, the rOpenSci project has established a code review system for scientific software that interfaces with the publication processes of the *Journal of Open-Source Software* and *Methods in Ecology and Evolution* (Ram et al., 2019).

2.7 | Code testing

Beyond reviewing the source code of a model, it is however also important to test its actual functioning (Table 1). In software engineering, this aspect of development is often divided into *unit testing* and *integration testing* (McConnell, 2004).

Unit tests target small, self-contained segments of the code, such as individual functions. They work by passing sample input to this segment and checking that the output conforms to expectations. This can be done manually (e.g. in a commandline environment), but ideally the developer writes a separate test function that is then saved alongside the actual code. Thus, an automatic test suite is accrued over time, which can be run periodically to ensure that later code changes did not introduce new bugs in the already existing code (this is called regression testing). Most programming languages have libraries either built-in or available that make creating such tests much easier. For examples of ecological software with a test suite, see the FENNEC tool and the DEBplant model (Table 2; Ankenbrand et al., 2018; Schouten et al., 2020).

A long-standing debate surrounds the extent to which such automated unit testing should be implemented. There are vocal proponents of a 'test-driven development' (TDD) approach, who argue that every function and feature in a code base should have an automated test accompanying it, and that this test should be written *before* the feature itself (e.g. Martin, 2009). As well as creating a comprehensive test suite for automated testing, such test-driven development encourages a more stringent and focussed programming style (Jeffries & Melnik, 2007). However, TDD is very time-consuming to implement, and in terms of defect-finding, other techniques such as code reviews appear to be more effective (see Section 2.6). Therefore, other authors like McConnell (2004) and Balaban et al. (2021) suggest using automated testing primarily for fundamental or critical functionality.

What must not be neglected in any case is integration testing. This refers to the testing of the complete software, including all submodules. This is somewhat more difficult to automate fully, although an automated integration pipeline greatly speeds up the process (see Section 2.8). Also, it may be possible to automate the comparison of certain output variables against the values of these variables as produced by previous model versions. Another classical technique of integration testing for IBMs is output visualisation (Ropella et al., 2002). Having the model display maps of its various entities and graphs of key metrics (such as population size or species number) will rapidly show patterns that run counter to or align with theoretical expectations (Grimm & Railsback, 2011).

Additionally, one can use logging facilities (see Section 2.5) or dedicated tools known as debuggers to inspect the state of internal model objects during its execution. These latter techniques are commonly used during debugging, that is, when searching for the root cause of a known defect. However, they can also be used proactively to investigate whether the software's internal working conforms to expectations, and thus to find bugs that may not be immediately visible in the model output. For more details, see the discussion on testing in (Grimm & Railsback, 2005, ch. 8.5).

2.8 | Automated integration

In the context of software development, integration refers to the combining of individual components into a larger system, and the subsequent testing of this system (McConnell, 2004). This is a regular procedure in an IBM workflow, where models do not just have multiple components in and of themselves, but may also have auxiliary software to set up and analyse experiments (see Section 2.2).

This process should be automated as much as possible, to avoid unnecessary complexity (Table 1). Every step that the user has to execute by hand is a step that may be forgotten, or done wrongly, or that at the very least takes up development time. Ideally, therefore, every fundamental action (like compiling and testing the source code, running and analysing an experiment, or packaging for release) should only require a single click or command. For example, in a newer version of the GeMM model (Table 2; Leidinger et al., 2021), we have three scripts to cover the complete modelling pipeline. The first script takes GeoTIFF files and converts them into the model's text-based landscape input format (see Section 2.3). The second script generates the set of configuration files needed for an experiment and launches the simulation runs. Finally, the third script reads in all output files and performs data analyses and visualisation.

In the software industry, this automation of the deployment pipeline has become known as continuous integration (CI, see the discussion in Brown & Wilson, 2011). It allows a faster development cycle, due to the smaller time gap between code development and execution. Also, adding automated testing into this pipeline means that bugs are discovered sooner rather than later, increasing developer confidence in their code (Hilton et al., 2016). All major git hosting platforms now support CI so that automated tests can be set up to run regularly, such as when merging a new feature branch. Considering IBMs, application of CI principles should furthermore reduce the usage complexity of models and help establish a greater reproducibility of results.

3 | NEXT STEPS

Having recognised the problem of double complexity inherent in IBMs, where should we go from here? As authors, we believe that we need a thorough discussion in the ecological modelling community about which complexity-containing techniques are applicable and feasible for our model software. This paper aims to continue and promote this discussion. Hopefully, the next few years will see more of these techniques integrated into our workflows, and explained in our textbooks.

Encouragingly, the field of climate modelling has already been through this very process. For example, Easterbrook and Johns (2009) described how one climate research centre developed their software, and contrasted this with practices in industry. Later, Rugaber et al. (2011) reviewed techniques for controlling complexity in coupled climate models. And more recently, Lawrence et al. (2018) discussed how developments in computer hardware and software engineering could be accommodated by the climate modelling community.

Of course, the institutional problem remains that most ecological modellers are biologists by training, not software developers. Facing this challenge at our institute, we teach multiple programming courses for both graduates and undergraduates (cf. Farrell & Carey, 2018). We have also discovered that collaborations with our university's computer science department can be very fruitful. We have heard talks from their professors, hosted their students for internships and organised joint courses; all to mutual profit.

Other research groups have been able to raise the funds to hire a professional software developer to help with their model building, creating positions similar to those of traditional lab technicians. This is generally only feasible for larger groups, as salaries in the IT industry are very competitive and often beyond the budget of most smaller research groups. This is especially problematic as many universities and funding agencies are still reluctant to pay for such positions (Nowogrodzki, 2019). Hence, although the approach of hiring professionals should be encouraged, training ecologists in computational skills will remain important for the foreseeable future.

Fortunately, the wider scientific community is becoming increasingly aware of the importance of good software development skills. The Software Carpentry project is a volunteer-based initiative offering basic 2-day software training courses for scientists (Wilson, 2016). Similarly, the rOpenSci community provides programming guidance and teaching, as well as building and maintaining a curated collection of software tools for ecological and evolutionary research (Boettiger et al., 2015). Complementing this, the concept of Research Software Engineering represents a push to create high-quality scientific software through close collaborations of scientists with professional software developers (Cohen et al., 2021).

As ecological modellers, we can profit from all of these initiatives. As a field, we need to cultivate a greater software engineering know-how, but this is not impossible. There is an extensive literature available on the topic, we can learn from colleagues in other departments, and, if funds permit, we can hire professionals to help us. Most importantly, we need to share experiences and develop a set of tried-and-tested best practices for our field.

4 | CONCLUSIONS

In this commentary, we have argued that we need to think about both biological and technical complexities if we want to get reliable results from our individual-based models. Although a lot of work has been done on biological model complexity, little has been written about the aspect of technical complexity. Fortunately, computer science has decades of experience in dealing with software complexity that we can learn from.

Key strategies are to avoid, subdivide, document and review complexity. Techniques to do so include writing clean code, choosing suitable languages and file formats, encapsulating submodules and abstraction layers, and documenting both biological and technical details. Additionally, code reviews and unit and integration tests are needed to verify code quality. Automated integration can speed up the modelling workflow and decreases the likelihood of procedural mistakes.

As IBMs continue to grow in scope and importance, learning to cope with their double complexity becomes increasingly vital. This paper provides some pointers, but we need more cross-pollination from the computer sciences, and a more thorough methodological discussion in the modelling community.

ACKNOWLEDGEMENTS

We are grateful to our colleagues at the CCTB for the years we have spent working and learning together as computational biologists. Thank you for your camaraderie! We also appreciate the comments by Bob O'Hara and two anonymous reviewers, which much improved this manuscript. M.A. and J.S.C. acknowledge funding from the 'Zentrum für digitales Experimentieren 4.0' (ESF-ZDEX). J.S.C. further acknowledges funding from the Bavarian Ministry of Science and the Arts in the context of the Bavarian Climate Research Network (bayklif).

CONFLICT OF INTEREST

The authors declare no conflict of interests.

AUTHORS' CONTRIBUTIONS

D.V. conceptualised the paper and wrote the original manuscript draft; M.A. contributed to the sections on language choice, code testing and automated integration; J.S.C. contributed to the list of example models, introduction and next steps sections, as well as providing supervision and securing funding. All authors were involved in review and editing.

PEER REVIEW

The peer review history for this article is available at https://publons.com/publon/10.1111/2041-210X.13716.

DATA AVAILABILITY STATEMENT

The sample code inspection checklist may be downloaded at https://doi.org/10.5281/zenodo.5284378 (Vedder, 2019).

ORCID

Daniel Vedder http://orcid.org/0000-0002-0386-9102
Markus Ankenbrand https://orcid.org/0000-0002-6620-807X
Juliano Sarmento Cabral https://orcid.org/0000-0002-0116-220X

REFERENCES

Abelson, H., Sussman, G. J., & Sussman, J. (1996). *Structure and interpretation of computer programs* (2nd ed.). Electrical Engineering and Computer Science Series. MIT Press.

Ankenbrand, M. J., Hohlfeld, S. C. Y., Weber, L., Förster, F., & Keller, A. (2018). Functional exploration of natural networks and ecological communities. *Methods in Ecology and Evolution*, *9*(9), 2028–2033.

Augusiak, J., Van den Brink, P. J., & Grimm, V. (2014). Merging validation and evaluation of ecological models to 'evaludation': A review of terminology and a practical approach. *Ecological Modelling*, *280*, 117–128.

Aurum, A., Petersson, H., & Wohlin, C. (2002). State-of-the-art: Software inspections after 25 years. *Software Testing, Verification and Reliability*, *12*(3), 133–154.

Balaban, G., Grytten, I., Rand, K. D., Scheffer, L., & Sandve, G. K. (2021). Ten simple rules for quick and dirty scientific programming. *PLoS Computational Biology*, *17*(3), e1008549.

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98.

Bocedi, G., Palmer, S. C. F., Pe'er, G., Heikkinen, R. K., Matsinos, Y. G., Watts, K., & Travis, J. M. J. (2014). RangeShifter: A platform for modelling spatial eco-evolutionary dynamics and species' responses to environmental changes. *Methods in Ecology and Evolution*, *5*(4), 388–396.

Boettiger, C., Chamberlain, S., Hart, E., & Ram, K. (2015). Building software, building community: Lessons from the rOpenSci Project. *Journal of Open Research Software*, *3*(1), e8.

Breloff, T. (2021). Plots.jl (v1.16.3). *Zenodo*. https://doi.org/10.5281/zenodo.4907285

Brooks, F. (1986). No silver bullet – Essence and accident in software engineering. In H.-J. Kugler (Ed.), *Proceedings of the IFIP Tenth World Computing Conference* (pp. 1069–1076). Elsevier Science B.V.

Brown, A., & Wilson, G. (Eds.). (2011). *The architecture of open source applications*. Creative Commons.

Cabral, J. S., & Kreft, H. (2012). Linking ecological niche, community ecology and biogeography: Insights from a mechanistic niche model. *Journal of Biogeography*, *39*(12), 2212–2224.

Cabral, J. S., Valente, L., & Hartig, F. (2017). Mechanistic simulation models in macroecology and biogeography: State-of-art and prospects. *Ecography*, *40*(2), 267–280.

Cabral, J. S., Whittaker, R. J., Wiegand, K., & Kreft, H. (2019). Assessing predicted isolation effects from the general dynamic model of island biogeography with an eco-evolutionary model for plants. *Journal of Biogeography*, *46*(7), 1569–1581.

Chi, F., Kurth, W., & Streit, K. (2016). Generating 3D models from a single 2D digitized photo using GIS and GroIMP. In *2016 IEEE international conference on functional-structural plant growth modeling, simulation, visualization and applications (FSPMA)* (pp. 22–27). IEEE.

Cohen, J., Katz, D. S., Barker, M., Hong, N. C., Haines, R., & Jay, C. (2021). The four pillars of research software engineering. *IEEE Software*, *38*(1), 97–105.

Cooper, N., & Hsing, P.-Y. (Eds.). (2017). *Reproducible code. BES guides to better science*. British Ecological Society.

Datseris, G., Vahdati, A. R., & DuBois, T. C. (2021). Agents.jl: A performant and feature-full agent based modelling software of minimal code complexity. arXiv:2101.10072 [nlin].

DeAngelis, D. L., & Grimm, V. (2014). Individual-based models in ecology after four decades. *F1000prime Reports*, *6*(June), 39.

Dijkstra, E. W. (1972). The humble programmer. *Communications of the ACM*, *15*(10), 859–866.

Easterbrook, S. M., & Johns, T. C. (2009). Engineering the software for understanding climate change. *Computing in Science Engineering*, *11*(6), 65–74.

Eklund, A., Nichols, T. E., & Knutsson, H. (2016). Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates. *Proceedings of the National Academy of Sciences of the United States of America*, *113*(28), 7900–7905.

Evans, M. R., Grimm, V., Johst, K., Knuuttila, T., de Langhe, R., Lessells, C. M., Merz, M., O'Malley, M. A., Orzack, S. H., Weisberg, M., Wilkinson, D. J., Wolkenhauer, O., & Benton, T. G. (2013). Do simple models lead to generality in ecology? *Trends in Ecology & Evolution*, *28*(10), 578–583.

Evers, J. B., & Bastiaans, L. (2016). Quantifying the effect of crop spatial arrangement on weed suppression using functional-structural plant modelling. *Journal of Plant Research*, *129*(3), 339–351.

Farrell, K. J., & Carey, C. C. (2018). Power, pitfalls, and potential for integrating computational literacy into undergraduate ecology courses. *Ecology and Evolution*, *8*(16), 7744–7751.

Grimm, V., Augusiak, J., Focks, A., Frank, B. M., Gabsi, F., Johnston, A. S., Liu, C., Martin, B. T., Meli, M., Radchuk, V., Thorbek, P., & Railsback, S. F. (2014). Towards better modelling and decision support: Documenting model development, testing, and analysis using TRACE. *Ecological Modelling*, *280*, 129–139.

Grimm, V., & Railsback, S. F. (2005). *Individual-based modeling and ecology*. Princeton University Press.

Grimm, V., & Railsback, S. F. (2011). Pattern-oriented modelling: A multi-scope' for predictive systems ecology. *Philosophical Transactions of the Royal Society B: Biological Sciences*, *367*(1586), 298–310.

Grimm, V., Railsback, S. F., Vincenot, C. E., Berger, U., Gallagher, C., DeAngelis, D. L., Edmonds, B., Ge, J., Giske, J., Groeneveld, J., Johnston, A. S. A., Milles, A., Nabe-Nielsen, J., Polhill, J. G., Radchuk, V., Rohwäder, M.-S., Stillman, R. A., Thiele, J. C., & Ayllón, D. (2020).

The ODD protocol for describing agent-based and other simulation models: A second update to improve clarity, replication, and structural realism. *Journal of Artificial Societies and Social Simulation*, *23*(2), 7.

Hagen, O., Flück, B., Fopp, F., Cabral, J. S., Hartig, F., Pontarp, M., Rangel, T. F., & Pellissier, L. (2021). Gen3sis: The general engine for eco-evolutionary simulations on the origins of biodiversity. *bioRxiv*, 2021.03.24.436109.

Harfoot, M. B. J., Newbold, T., Tittensor, D. P., Emmott, S., Hutton, J., Lyutsarev, V., Smith, M. J., Scharlemann, J. P. W., & Purves, D. W. (2014). Emergent global patterns of ecosystem structure and function from a mechanistic general ecosystem model. *PLOS Biology*, *12*(4), e1001841.

Hatton, L. (1997). The T-experiments: Errors in scientific software. In R. F. Boisvert (Ed.), *Quality of numerical software: Assessment and enhancement, IFIP advances in information and communication technology* (pp. 12–31). Springer US.

Hemmerling, R., Kniemeyer, O., Lanwert, D., Kurth, W., & Buck-Sorlin, G. (2008). The rule-based language XL and the modelling environment GroIMP illustrated with simulated tree competition. *Functional Plant Biology*, *35*(10), 739.

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM international conference on automated software engineering (ASE)* (pp. 426–437). ACM.

Hoeks, S., Tucker, M. A., Huijbregts, M. A. J., Harfoot, M. B. J., Bithell, M., & Santini, L. (2021). MadingleyR: An R package for mechanistic ecosystem modelling. *Global Ecology and Biogeography*, *30*(9), 1922–1933.

Jeffries, R., & Melnik, G. (2007). Guest editors' introduction: TDD–The art of fearless programming. *IEEE Software*, *24*(3), 24–30.

Lawrence, B. N., Rezny, M., Budich, R., Bauer, P., Behrens, J., Carter, M., Deconinck, W., Ford, R., Maynard, C., Mullerworth, S., Osuna, C., Porter, A., Serradell, K., Valcke, S., Wedi, N., & Wilson, S. (2018). Crossing the chasm: How to develop weather and climate models for next generation computers? *Geoscientific Model Development*, *11*(5), 1799–1821.

Lee, B. D. (2018). Ten simple rules for documenting scientific software. *PLOS Computational Biology*, *14*(12), e1006561.

Leidinger, L., Vedder, D., & Cabral, J. S. (2021). Temporal environmental variation may impose differential selection on both genomic and ecological traits. *Oikos*, *130*(7), 1100–1115.

Lorscheid, I., & Meyer, M. (2016). Divide and conquer: Configuring submodels for valid and efficient analyses of complex simulation models. *Ecological Modelling*, *326*, 152–161.

Lubin, M., & Dunning, I. (2015). Computing in operations research using Julia. *INFORMS Journal on Computing*, *27*(2), 238–248.

Manson, S., An, L., Clarke, K. C., Heppenstall, A., Koch, J., Krzyzanowski, B., Morgan, F., O'Sullivan, D., Runck, B. C., Shook, E., & Tesfatsion, L. (2020). Methodological issues of spatial agent-based models. *Journal of Artificial Societies and Social Simulation*, *23*(1), 3. https://doi.org/10.18564/jasss.4174

Martin, R. C. (Ed.). (2009). *Clean code: A handbook of agile software craftsmanship*. Prentice Hall.

McConnell, S. (2004). *Code complete* (2nd ed.). Microsoft Press.

Nowogrodzki, A. (2019). Tips for open-source software support. *Nature*, *571*(7763), 133–134.

Parry, H. R. (2009). Agent-based modeling, large-scale simulations. In M. Sotomayor, D. Pérez-Castrillo, & F. Castiglione (Eds.), *Complex social and behavioral systems* (pp. 913–926). Springer US.

Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. D. V., Fufezan, C., Ternent, T., Eglen, S. J., Katz, D. S., Pollard, T. J., Konovalov, A., Flight, R. M., Blin, K., & Vizcaíno, J. A. (2016). Ten simple rules for taking advantage of Git and GitHub. *PLOS Computational Biology*, *12*(7), e1004947.

Petter, G., Zotz, G., Kreft, H., & Cabral, J. S. (2021). Agent-based modeling of the effects of forest dynamics, selective logging, and fragment size on epiphyte communities. *Ecology and Evolution*, *11*(6), 2937–2951.

Railsback, S., Ayllón, D., Berger, U., Grimm, V., Lytinen, S., Sheppard, C., & Thiele, J. (2017). Improving execution speed of models implemented in NetLogo. *Journal of Artificial Societies and Social Simulation*, *20*(1), 3.

Ram, K., Boettiger, C., Chamberlain, S., Ross, N., Salmon, M., & Butland, S. (2019). A community of practice around peer review for long-term research software sustainability. *Computing in Science Engineering*, *21*(2), 59–65.

Rangel, T. F., Edwards, N. R., Holden, P. B., Diniz-Filho, J. A. F., Gosling, W. D., Coelho, M. T. P., Cassemiro, F. A. S., Rahbek, C., & Colwell, R. K. (2018). Modeling the ecology and evolution of biodiversity: Biogeographical cradles, museums, and graves. *Science*, *361*(6399), eaar5452.

Raymond, E. S. (2003). *The art of UNIX programming*. Addison-Wesley professional computing series. Addison-Wesley.

Rigby, P. C., & Bird, C. (2013). Convergent contemporary software peer review practices. In B. Meyer, L. Baresi, & M. Mezini (Eds.), *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering – ESEC/FSE 2013* (pp. 202–212). ACM Press.

Ropella, G. E., Railsback, S. F., & Jackson, S. K. (2002). Software engineering considerations for individual-based models. *Natural Resource Modeling*, *15*(1), 5–22.

Rugaber, S., Dunlap, R., Mark, L., & Ansari, S. (2011). Managing software complexity and variability in coupled climate models. *IEEE Software*, *28*(6), 43–48.

Schiffers, K., Schurr, F. M., Travis, J. M. J., Duputié, A., Eckhart, V. M., Lavergne, S., McInerny, G., Moore, K. A., Pearman, P. B., Thuiller, W., Wüest, R. O., & Holt, R. D. (2014). Landscape structure and genetic architecture jointly impact rates of niche evolution. *Ecography*, *37*(12), 1218–1229.

Schouten, R., Vesk, P. A., & Kearney, M. R. (2020). Integrating dynamic plant growth models and microclimates for species distribution modelling. *Ecological Modelling*, *435*, 109262.

Shull, F., Basili, V., Boehm, B., Brown, A. W., Costa, P., Lindvall, M., Port, D., Rus, I., Tesoriero, R., & Zelkowitz, M. (2002). What we have learned about fighting defects. In *Proceedings Eighth IEEE Symposium on software metrics* (pp. 249–258). IEEE.

Sieger, C. S., & Hovestadt, T. (2020). The degree of spatial variation relative to temporal variation influences evolution of dispersal. *Oikos*, *129*(11), 1611–1622.

Stillman, R. A., Railsback, S. F., Giske, J., Berger, U., & Grimm, V. (2015). Making predictions in a changing world: The benefits of individual-based ecology. *BioScience*, *65*(2), 140–150.

Stodden, V., & Miguez, S. (2014). Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *Journal of Open Research Software*, *2*(1), e21.

Sun, Z., Lorscheid, I., Millington, J. D., Lauf, S., Magliocca, N. R., Groeneveld, J., Balbi, S., Nolzen, H., Müller, B., Schulze, J., & Buchmann, C. M. (2016). Simple or complicated agent-based models? A complicated issue. *Environmental Modelling & Software*, *86*, 56–67.

Vedder, D. (2019). Data from: CCTB code inspection checklist. *Zenodo*, https://doi.org/10.5281/zenodo.5284378

Vedder, D., Leidinger, L., & Cabral, J. S. (2020). Effects of species traits and abiotic factors during the stages of plant invasions. *bioRxiv*, https://doi.org/10.1101/2020.04.20.050278

Wilson, G. (2016). Software Carpentry: lessons learned. *F1000Research*, *3*, 62–https://doi.org/10.12688/f1000research.3-62.v2

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M.

D., Waugh, B., White, E. P., & Wilson, P. (2014). Best practices for scientific computing. *PLoS Biology*, *12*(1), 1–7.

Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T. K. (2017). Good enough practices in scientific computing. *PLoS Computational Biology*, *13*(6), e1005510.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2018). Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, *44*(10), 951–976.

How to cite this article: Vedder, D., Ankenbrand, M., & Sarmento Cabral, J. (2021). Dealing with software complexity in individual-based models. *Methods in Ecology and Evolution*, 00, 1–10. https://doi.org/10.1111/2041-210X.13716