Stuart Rosser (2015)

Simulation Design: Trans-Paradigm Best-Practice from Software Engineering

Journal of Artificial Societies and Social Simulation 18 (3) 9

Abstract

There are growing initiatives to apply software engineering (SE) best-practice to computational science, which includes simulation. One area where the simulation literature appears to be particularly light is in the overall structural design of simulations, and what architectures and features are valuable for what reasons. (Part of the problem is that parts of this knowledge are abstracted away in simulation toolkits which are often not easily comparable, and have different conceptual aims.) To address this, I outline three key software properties which embody SE best-practices, and then define an ‘idealised’ software architecture for simulation—what SE would call a reference architecture—which strongly exhibits them. I show that this is universal to all simulations (largely because modelling-paradigm-specific detail is encapsulated into a ‘single black box’ layer of functionality) but that simulation toolkits tend to differ in how they map to them; this relates to the aims of the toolkits, which I provide a useful categorisation of. I show that, interestingly, there are also some features of this architecture that are not fully represented in any simulation toolkit that I am aware of. I present a library—JST—which provides some proof-of-concept implementations of them for Java-based toolkits. This library, and other ideas in the reference architecture, are put into practice on a published, multi-paradigm model of health and social care which uses the AnyLogic toolkit. I conclude with some thoughts on why this area receives so little focus, how to take it forwards, and some of the related cultural issues.

Keywords:
Software-Engineering, Simulation-Toolkits, Reference-Architecture, Best-Practice

Introduction

1.1 As much of science becomes more and more dependent on software, there has been increasing interest in promoting the development of scientific software which is reliable (well-tested), reusable, well-maintained (sustainable), and can be used in ways which provides open, reproducible research. Such initiatives have been championed by groups such as the UK’s research-council-funded Software Sustainability Institute[1], the Science Code Manifesto[2] and related authors (Stodden et al. 2010; 2013); and the Software Carpenter movement[3] (Wilson et al. 2014; Wilson 2014). By definition, these aims involve selectively applying software engineering (SE) best-practice ideas to the development of scientific software, including simulation.

1.2 If we restrict ourselves to simulation software, and to simulation design (ignoring aspects of development process and code access), there are three main strongly-related areas of best-practice which I would regard as universal to all simulation (precisely because they are universal to all software), and which are echoed in computational science best-practice papers (Sandve et al. 2013; Wilson et al. 2014), software engineering textbooks (Sommerville 2011), and practitioner best-practice handbooks (McCormick 2004), all backed by empirical research (Oram & Wilson 2010)[4].

Automated Reproducibility

Being able to recreate any run of the software—for testing purposes and to check claims about its outputs—in an automated way (not just via manual recreation from documentation). This involves provenance (and perhaps automated recreation) of the entire computational environment (since results can vary based on things like the versions of external libraries used).

Cohesive, Loosely-Coupled Design

A design separated into components with well-defined (cohesive) functions, and minimised dependencies on other components (loose-coupling). This massively aids the debugging, maintainance and reusability of the code. This often involves reusing recurring structural and behavioural forms that have been shown to help solve common design issues: SE calls such forms design patterns (Gamma et al. 1995; Buschmann 1996). Such forms help establish a shared software design vocabulary at a higher level of abstraction.

Testability

Being designed in a way that facilitates testing at different levels (e.g., single class, component or whole system) and, where possible, includes automated tests as part of the software deliverable. In particular, automated tests provide a bank of regression tests which can be continually run to check that changes have not caused bugs elsewhere (i.e., caused previously successful tests to fail). Such tests become the central driver of the development process in the increasingly-used Test-Driven Development (TDD) approach (Jeffries & Malik 2007).

1.3 Software exhibiting these properties is highly reusable (given access to it) and its implementation will typically involve reuse of existing software where it exists. In the simulation domain, there are many toolkits which provide reusable, well-tested software for simulation development which include (a) templates for model elements relating to one or more modelling paradigms—such as agent-based modelling (ABM), discrete-event simulation (DES) or system dynamics (SD); and (b) supporting infrastructure code to create and run models, such as for visualisations, simulation control interfaces, and input/output handling.

1.4 There has been some emerging work which tries to define new simulation frameworks and abstractions which better embody some of these principles; e.g., the modular architecture and best-practices of JAMES II (Uhrmacher 2012; Himmelspach & Uhrmacher 2007), or test and experiment specifications which are model-based (Demaille et al. 2011) or domain-language-based (Ewald & Uhrmacher 2014)[5].

However, there appears to be virtually no discussion of these issues more generally for ‘mainstream’ simulation using widely-used toolkits such as, in the ABM case, NetLogo (Tisue & Wilensky 2004), Repast (North et al. 2013), MASON (Luke et al. 2009), or AnyLogic (Borshchev & Filippov 2004). In particular, there is nothing which allows simulation practitioners to understand how these ideas might be embodied in some best-practice simulation design, and to therefore have some frame to assess existing toolkits and make more informed decisions on their choice of simulation platform (and thus understand the strengths and weaknesses of their simulation software design with respect to this best-practice)[6].

1.5 If we restrict ourselves to the ABM domain for now, literature which does not really discuss this best-practice includes (a) toolboxes (Rabbiabz & Grimm 2012; Gilbert & Troitzsch 2005; Miller & Page 2007; Grimm & Railsback 2005); (b) toolkit comparisons (Alan 2010; Nicolci & Milkay 2009; Railsback et al. 2009); (c) toolkit description papers (Tisue & Wilensky 2004; Luke et al. 2009; North et al. 2013; Borshchev & Filippov 2004); and (d) best-practice papers (Ropella et al. 2002; North & Macal 2006; Grimm & Railsback (2005); (d) and North & Macal (2014)) get closest, mentioning such things as automated testing, design patterns and separation of model from visualisation. However, none of these relates to concrete design principles. Toolkit description papers discuss some features which embody some of these best-practices, but do not explicitly make the connections or discuss alternatives. There are also a few recent papers focusing on automated testing and TDD (Gürçan et al. 2013; Collier & Dzik 2013), but not on how to architect simulations for testability.[7]

1.6 Outside of ABM (but from a less rigorous exploration), it appears to be a similar situation. The practical operational research literature (which includes some of the ABM references above) is reasonably interested in simulation software development but, again, typically in process- and paradigm-specific conceptual design.

Aims

1.7 This paper aims to do several things: (1) draw attention to this gap and provide (indirectly) a rough guide to relevant SE and simulation literature; (2) provide a frame which captures some best-practice simulation architecture and can be used to generally understand how simulations are to be constructed (section 2); (3) use this frame for a better understanding of simulation toolkits and, in particular, highlight some consistent omissions (section 3); (4) develop a software library which begins to address the omissions for a broad family of toolkits (section 4); and (5) provide a case study where the frame and library are used on a published, multi-paradigm simulation model using the AnyLogic toolkit (section 5), where AnyLogic was chosen for particular reasons.

1.8 To make these ideas relevant for all (dynamic) simulations, the abstraction of the real-world system is cleanly separated from the rest of the simulation (see section 5) and not ‘opened up’; we shall see that there is still lots of detail to work with in the remainder.

1.9 In the final conclusions (section 6), I draw back to consider why this knowledge gap seems to exist, and some of the more general cultural barriers in applying this SE-oriented thinking. I also discuss where this work fits with respect to the emerging ideas mentioned earlier. Although I believe that the ideas here are largely universal, I focus mostly on ABM modeling in this paper, so the conclusions also reflect briefly on this broader applicability.

A Best-Practice Paradigm-Independent Frame

2.1 There are some recurring, generic SE design patterns which are known to strongly assist in creating software which exhibits the properties in section 1; these still need applying to a simulation domain using domain understanding, and many interpretations are possible. Equally, there are also some simulation-specific features which I believe are also important for simulations to exhibit the best-practice properties, and are less directly related to generic SE design patterns.
Layered Functionality

2.2 I develop these ideas to create a kind of ‘idealised’ software architecture for simulations. In SE terms, this is known as a reference architecture: “[they] capture important features of system architectures in a domain. Essentially, they include everything that might be in an application architecture [...] The main purpose [...] is to evaluate and compare design proposals, and to educate people about architectural characteristics in that domain” (Sommerville 2011, §6.4). ’Software architecture’ is a slightly contentious term, but I like Fowler’s take (Fowler et al. 2003, p.1):

“Architecture’ is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, [design] decisions that are hard to change.”

2.3 The second part of Fowler’s quote is important here: modellers should think about these issues up-front, because it is difficult to retrospectively re-design a model to match this architecture and, perhaps more relevantly, the choice of simulation toolkit will affect how possible this is and how much work is involved. Toolkits typically also present themselves conceptually in a particular way, dependent on their design goals, which will cause differing degrees of ‘dissonance’ with the reference architecture even if, underneath, the toolkit code is directly mappable to it.R

2.4 The reference architecture is cumulatively built-up in the sub-sections which follow (and in figures 1–3). We will see concrete examples of the architecture in practice in sections 3 and 5.

Layered Functionality

2.5 A key SE best-practice is to separate domain model code—the code which represents the concepts of the domain which the software relates to (e.g., banking, aircraft-control)—from non-domain-model code as part of a layered architecture (Evans 2004; Fowler et al. 2003): functionality is split into layers, where each layer has a well-defined role and is dependent only on layers ‘below’ it. This directly forms a cohesive, loosely-coupled design, but also aids testability (in more easily being able to compose parts of the application needed for different levels of testing) and automated reproducibility (in terms of isolating the core part—the domain model—whose behaviour needs to be reproduced).

2.6 In the case of simulation, the domain model is the abstraction of the relevant real-world system (simuland) as some set of behavioural entities acting and interacting over simulated time, each with their own state. This thus includes the overall handling of space and time. (Time, for example, is typically handled via fixed time steps or a discrete-event schedule.) Modeling paradigms are encapsulated in this layer: they define what these entities can or should be, how they may act or interact, and how space and time are represented.

2.7 How this architecture works is best understood by describing the full set of layers from the bottom upwards, summarised in figure 1. (I do not always explicitly state it, but the descriptions should make clear how each layer will only need the services of layers beneath it.)

![Layered Functionality Diagram](http://jasss.soc.surrey.ac.uk/18/3/9.html)

**Figure:** Layered functionality for simulation code, which defines a generic layered architecture. This is extended in figures 2 and 3.
In particular, note that a given domain model might have multiple presentations, with multiple alternative visualisations per component; such solutions require a layered domain model separation.

Experiments Definition. The parts of the user interface which support the definition of simulation runs (experiments), possibly including multi-run experiments. This mainly consists of how model inputs are defined and passed on to the model, and any automated manipulation of them across multiple runs for things like sensitivity analysis. Because this tends to be particularly generic to any simulation (and modellers using multiple toolkits may want a vendor-neutral solution), separate experimental platforms exist (Guliyas et al. 2011), and I am aware of simulation consultancies who develop their own in-house.

Relevant Cross-Cutting Concerns

2.8 There are some capabilities that require code across most or all layers, i.e., the functionality cannot be isolated into a single layer. In SE, these are called cross-cutting concerns. A good example outside of our reference architecture is the capability to freeze' simulation state to file, and `re-neutralize' a simulation later to run from that point, which many simulation toolkits support. This typically relies on serialisation technologies in the underlying programming language, but requires that all the objects that are going to be serialised are coded in a particular way. In some cases, one might just want to freeze the state of the domain model but, in others, there might also be a need to capture the state of the model presentation (so higher layers also need to be coded to support serialisation).

![Figure 2: Figure 1, with three important cross-cutting concerns added.](http://jasss.soc.surrey.ac.uk/18/3/9.html)

**KEY**
- Domain-model layer
- Non-domain-model layer
- Cross-cutting capability

2.9 In relation to our three best-practice properties, there are three specific cross-cutting concerns that need adding to our reference architecture (see figure 2):

- Logging. Logging frameworks are a common tool in virtually all programming languages. They allow messages to be logged from applications, and support useful features such as multiple levels of logging detail which can be switched to as needed (including turning on more detailed logging only for certain areas of code); separation of messages per thread (for multi-threaded applications); and the automatic addition of fields (such as timestamp headers). Even where a debugger is available, the 'global' (full history of the run) and user-defined nature of log messages makes them complementary and perceived by many as preferable: 'The most effective debugging tool is still careful thought, coupled with judiciously placed print statements’ (Kernighan 1979). Such logs can also be used to automate model tests (see later).

In simulation terms, we typically want diagnostic logs of what the domain model is doing, with the ability to vary detail level per component at run-time. Such logs should be separated per simulation run, and with message headers useful for the simulation context (e.g., with the current simulated time in them). Even when simulation tools allow for visual, run-time navigation through the model and its state, this is still just an immediate snapshot view (compared to a full history in logs), has a fixed level of detail, and focuses on state, not (the algorithmic) details of entity behaviour.

- Event-Driven Separation. In a layered architecture, upper layers often need to know when particular events (in the general sense) occur in lower layers; the classic example being when a visualisation of some data needs to know when the data has changed (to update itself).[13] But lower layers should be unaware of the existence of upper layers (i.e., be independent of them in their operation). More generally, this applies when any object wants to know about changes in state of another object without the latter having to explicitly know about the former.

The normal design pattern to achieve this is to use a publish-subscribe design (Gamme et al. 1995, p.293), also known as an observer pattern. Objects publish events when they occur (to some events manager), and observers subscribe to events that they are interested in, being notified when they occur.[14]

In simulation terms, this is clearly useful in separating the State & Control Presentation and Meta-Data Capture layers from the Domain Model one in certain circumstances. (If some meta-data is captured daily, then the capture process can run only daily and derive the state directly; no publish-subscribe design is needed. Compare that with needing to display every time some specific random change happens to a model entity.)

However, this can also be very useful within the Domain Model layer. Most state changes that happen there have meaning in the domain model in terms of a causal event, e.g., ‘Agent A said 6 widgets to Agent B on market day 1’.[14] In this example, other agents might be interested in what transactions occurred so as to change their trading strategies for future days. We do not want each agent to explicitly call every other agent to communicate this; the publish-subscribe design is a much cleaner solution. This is also advantageous in aligning the model implementation with its conceptual design: often we are modeling the ‘receipt/mission of such information (possibly with imperfect transmission) in real-life, which is much more well-represented in the publish-subscribe design (given that we are abstracting this transmission process).

In fact, these domain events are also typically the events that the upper layers are interested in anyway: we want to present or output about the domain-contextual ‘things that are happening’. Thus, this is not typically end up publishing two different ‘styles’ of event. Finally, the domain events (perhaps with some filtering) naturally provide a narrative for what is happening in the model which is very useful for communicating with a model audience, in testing, and in understanding how a model achieves the outputs it does (Millington et al. 2012). Outputting such a narrative naturally combines with the previous logging features (as another type of log to diagnostic ones).

Visual Interactive Execution. Being able to observe the simulation unfolding in a suitable graphical user interface (GUI) is very useful for detecting both coding and conceptual errors (Grim & Railsback 2005, §8.5.1), and thus enhancing testability. This does not have to be a cross-cutting concern if this view is just ‘read-only’. However, what is additionally useful is to be able to interactively interact with the model at run-time, typically to change model parameters or state on-the-fly so as to be able to experiment more dynamically (often used by information from the model visualisation). This is a form of computational steering for model exploration, though that term is normally reserved for large-scale mathematical computations (Mudal et al. 1999, §1). This does require support across the layers since, for example, domain model components should expose their state in a given way so that there exist built-in visualisations that allow them to be changed; the components also need to code how they react to the change, depending on the function of the state or model parameter changed.

Some Layer-Specific Features

2.10 To complete our full reference architecture (figure 3), there are two simulation-specific features which sit in particular layers.
Run-Reproducibility Support.
If we want to be able to reproduce simulation runs in an automated way, we need features which can record (a) details of the model code (e.g., its location in a version control system—see Wilson et al. (2014)); (b) details of all domain-model-specific parameters; and (c) details of the environment (e.g., name and version of all toolkits used, Java virtual machine details, operating system). Ideally we also want features which can reconstitute the model and its environment from this data, though that is significantly harder to automate.

This naturally falls into the Meta-Data Capture layer.

Test-Oriented Stochasticity Control.
When testing or exploring a stochastic simulation, we often want to adjust the stochasticity so that we can ‘better see what’s going on’. For example, we might want to turn off randomness in some areas (typically by reverting to mean values) and/or accentuate it in others in the same functional direction so that we can see the effect of a particular sub-component (area of functionality) more clearly. In all cases, it is much more preferable if we can do this without having to manually change original model parameters or, worse, model code. (It is very common for particular stochastic elements to be hard-coded into the model, either for ease of coding or because the modeller does not expect users to need to change them. [15])

Thus, functionality which allows us to make these temporary adjustments external to the model code and parameters would be very useful. This requires support both at the Utilities layer (to have probability distributions coded in such a way that they support this dynamic stochasticity control) and at the Experiments Definition layer (to be able to specify the particular ‘overrides’ one wants for this run).

2.1.1 As mentioned at the start of this section, there are other possible reference architectures and, in particular, JAMES II (Himmelspach & Uhrmacher 2007) treats similar ground but for a different purpose. To set the reference architecture in better context, I make some brief comparisons in appendix A.

Understanding Simulation Toolkits
3.1 I want to show how the reference architecture can help provide insight into toolkit design by showing how some toolkits map to it. However, a particular toolkit classification is needed first to help set things in some useful context. (It is based on my own assessment of what makes most difference in the nature of the toolkit.) Figure 4 provides a visual summary for a number of popular toolkits: primarily ABM-oriented ones, though Simul8[16] is a DES-specific toolkit and AnyLogic supports ABM, DES and SD[17].

3.2 The axes represent two strong (and somewhat correlated) distinguishing ‘positions’:
Some toolkits, to a greater or lesser degree, permit code to be designed visually, typically by dragging and dropping configurable ‘widgets’ and (where appropriate) linking them together. Some form of textual code is normally required in addition to provide logic which cannot easily be represented visually, but a visually-oriented toolkit will normally attempt to minimise this. There is still considerable debate in computer science over when such visual programming languages provide benefits (both in terms of the type of code they are best for and what types of users most benefit), and when there are trade-offs (e.g., in the expressivity of the language). Authors such as Green & Petre (1996) and Whitley & Blackwell (2001) give a flavour of this.

Visual coding has long been the norm in commercial operational research (OR) orientated offerings, such as Simul8 for DES, (Think for SD[18], and the multi-paradigm tool AnyLogic. Elements of visual coding are
becoming more prominent in ABM—e.g., Repast Simphony’s visual statechart construction, and the ABM components of AnyLogic. When the code is designed visually, this also normally provides a run-time visualisation of the code as it executes, thus providing a particular fixed form of visualisation ‘for free’.

Although NetLogo is coded non-visually, it takes an alternative route towards conceptual abstraction: providing a paradigm-specific simulation language (as most DES toolkits historically did).

Library–vs. Integrated–Tool–Like.

Some toolkits exist ‘just’ as a set of core libraries which provide a framework for constructing simulations, such as MASON. Others integrate this framework together with related GUI tools to aid code construction and testing, such as those for debugging, version control, and code navigation. (The visual GUI nature of the IDE should not be confused with visual programming of the actual model components.) In general, such integrated tools are referred to as integrated development environments (IDEs) in SE.

When using a more library-like toolkit written in a general-purpose programming language, the modeller can use third-party general IDEs (such as Eclipse™ for Java). However, a simulation-toolkit-specific IDE will also typically integrate ‘helper’ tools tailored to the simulation-specific reusable components provided by the framework, and that simulations are used via experiments. Visual coding will tend to lead to this form of IDE, since most elements of it are needed to do the visual coding.

This continuum normally goes hand-in-hand with another distinction: integrated tools tend to be object-based rather than object-oriented (Jones & Roberts 1999, §4). Whilst all modern toolkits are written in an object-oriented programming language, an object-based one presents an abstraction to the user of a fixed set of paradigm-specific components (or, in NetLogo’s case, a fixed syntax), which can only be extended via composition. An object-oriented one exposes a set of classes that can be extended via standard object-oriented techniques, which is typically more flexible at the expense of complexity.

Less formally, I think that the two ends of the spectrum are closely linked to cultures of software use, and reuse the terms user and programmer cultures used by Guo (2001). Library-like toolkits lean towards a programmer culture, which focuses on the qualities of good software exposed by SE. Thus, things like expressive power, flexibility and reusability are important. There is also an understanding that best-of-breed external libraries would be used as needed by the programmer, with the flexibility in this choice outweighing the need to pick one and strongly integrate it into the toolkit. Integrated-tool-like toolkits lean towards a user culture, where the focus is on providing integrated tools for the task-in-hand. Such tools typically provide their own conceptual models which try to hide the underlying implementation complexity and provide convenient high-level abstractions (which also steer the user towards a particular way of thinking about the task). In simulation terms, this is also related to ideas that domain experts (scientists) should be able to create models without also being programmers (Borshchev & Filippov 2004, §6), and that SE skills are potentially too difficult, intimidating and/or time-consuming to learn (Brailsford 2014, §2).

Both types of toolkit may also support extensibility via other mechanisms: either ‘hook points’ where snippets in a textual programming language can be inserted (e.g., Simul8™) or being able to extend the syntax of the domain-specific language via lower-level code (e.g., NetLogo). AnyLogic is a good example of a ‘halfway house’ which provides an object-oriented platform (but with some restrictions) and lots of object-based components with configuration and extensibility via Java snippets.

3.3 We will now have a look at how MASON and AnyLogic map to the layered aspects of our reference architecture. (We will come on to the cross-cutting concerns for all toolkits afterwards.) This is a good pair to choose because they are relatively far apart in our classification, whilst both still being object-oriented (or largely so in AnyLogic’s case) which tends to expose the internal code architecture more clearly. In both cases, I do not explain the exact function of all the components shown for space purposes; where they are not self-explanatory, the interested reader can look them up in the toolkits’ documentation (which is publicly available for both toolkits).

MASON Reference Architecture Mapping

3.4 The mapping is summarised in figure 5. There are three main things to notice.

![Figure 5: How core components of the MASON toolkit map onto the reference architecture. The dotted border sections of some multi-layer components indicate layers that they do not cover. An RNG is a random number generator.](http://jasss.soc.surrey.ac.uk/18/3/9.html)
Firstly, despite being very library-like and ‘programmer-friendly’, MASON still diverges from the reference architecture in some areas (see the three layer-spanning components). Having said that, these divergences are largely pragmatic decisions: (i) There is a sharp layered distinction between model and simulation (SimState and GUIState[22]), but meta-data capture components can just be included in one or the other, depending on whether the modeller considers them part of the model ‘core’ or not. If one thinks of the model as an input-output converter, it makes sense to put any meta-data capture used for writing permanent outputs into the core model. (ii) Steppable is a generalised interface for something that performs actions in the simulation, and so applies to any model component doing things in simulated time. However, domain model components are still strongly partitioned by being part of the SimState. (iii) The class-level (Java static) SimState code which spans two layers (the doLoop method) is really just a helper function to instantiate and launch models, running them a given number of times for a given time window. It would be overkill to separate that code into layers. (iv) Similarly, GUILite primarily covers the ‘thin’ Execution Control functionality, but also serves as a container for all the presentation components, as well as the domain model which it ‘wraps’.

Secondly, MASON gives the modeller transparent, fairly low-level access to the building blocks, particularly for model user interfaces. The main GOLl Console is augmented with user-defined displays which contain inspectors (agent state presentation), portraits (presentation of spatial or topological fields), or charts and plots. The underlying Java Swing GUILite components are ‘visible’ and accessible in many places.

Thirdly, though it is not clear from the figure, MASON does have a very clean separation of the Execution Control layer, in that there is a separate schedule for non-domain-model objects, with the Execution Control logic stepping through the domain-model schedule and, after all processing for a given simulation time, processing any on the non-domain-model schedule.

3.8 The mapping is summarised in figure 6.

3.9 Firstly notice how, compared to MASON, the components are more spread across the layers. This is primarily because all the domain model components (agent and DES/SD components) have built-in run-time visualisations, and thus they have ‘vertical’ functionality. This is misleading in the sense that user presentation code can still make the layered separation if it wishes to (though AnyLogic does not tend to encourage this), which we will see in the case study[20]. Since AnyLogic focuses more on being an integrated tool, it wants a lot of its components to provide ‘useful’ default functionality where the user does not typically make distinctions between model and visualisation. (There are also commerical reasons to make model components not reasable outside of the AnyLogic ecosystem.)

3.10 Secondly, unlike MASON, there is an explicit Meta-Data Capture layer component (Data Set). This is primarily to capture some standard functionality in a visually-developed widget: maintain a time series of real numbers (floating-point values), with an event to periodically populate it (where the ‘interleaving’ of this event with others is hidden from the user). In fact, AnyLogic merges the Execution Control and Domain Model layers in having non-domain-model objects sharing a master schedule with domain model ones. With the current implementation, this means that these meta-data capture events are not robust; they can end up occurring before domain model actions have finished at a given simulation time[23]. AnyLogic charts can also use this data type explicitly as the source for their visualisation.

3.11 MASON essentially allows the same thing, but the modeller creates the parts explicitly, using whatever data structure they want. (They may not want to capture a numeric value, for example.) An AnyLogic modeller can also take this approach if desired, since AnyLogic also exposes low-level Java features.

AnyLogic Experiment for every run that needs to be reproducible, and there is nothing to stop the user inadvertently changing such experiments.

3.13 For logging, SimU8 provides logging automatically, but only for its own components and with fixed content. Most tools provide the capability for user code to write messages to the console, but with no switchable detail level, no per-run separation for multi-run experiments, and no permanent file capture.[22]

3.14 For event-driven separation, features such as Repast Symphony ‘watchers’ and AnyLogic message passing can be used to do similar things. However, neither centralises the idea of domain events (with a related narrative), with both focusing on specifying senders (or sources) and receivers without decoupling them.

3.15 For run-reproducibility, some toolkits (e.g., AnyLogic) allow experiment settings to be defined and retained, so experiments can be reproduced by re-running with those saved settings. However, this captures nothing about the environment (e.g., the version of the toolkit used) and relies on user discipline and effort to maintain a proper traceable link between run and experiment. (Ideal, one should create a new AnyLogic experiment for every run that needs to be reproducible, and there is nothing to stop the user inadvertently changing such experiments.)

3.16 For task-oriented stochasticity control, AnyLogic effectively has probability distributions defined as separate object instances (unlike many other toolkits, including MASON and AnyLogic). Thus, these objects can be changed to a simpler distribution as needed for testing purposes (e.g., an exponential distribution with mean of 2 can be replaced with a fixed distribution returning 2 so as to ‘collapse the exponential to its mean’). However, this still requires the user to determine a relevant alternative distribution (i.e., the operation the user wants to perform—e.g., collapse to mean—is not explicit) and this still involves changing the ‘real’ parameters of the model (and remembering to change them back afterwards), rather than applying per-run override settings.

3.17 Even where the capabilities partially exist, the lack of consistency across toolkits is problematic. In terms of why these are not included, I think this is largely a question of toolkit focus and style (see section 3). They are not included in user-culture-oriented toolkits because they tend to present a programmer-oriented view of thinking about how to develop and test simulation software which is not consistent with the user-oriented view; instead, limited user-centric versions of them are sometimes included, and users who might want them would be expected to develop these facilities themselves. I expected them to be more likely to exist in programmer-culture-oriented toolkits, and I can only think they are missing because (a) the toolkits focus on the simulation core and not the overall modelling process (which these aspects strongly relate to); (b) they are areas where programmers might prefer the flexibility of defining their own approach—all the aspects have a few different ways to approach them (some more heavyweight than others), plus logging and event-driven separation are standard SE design patterns, with the former having widespread library implementations.

3.18 I do not think that any of these reasons are suitable justification not to look at developing standardised solutions for simulation, and there is considerable implementation complexity, even for less simulation-specific areas like logging where existing libraries provide much of the functionality.
The JSIT Library

4.1 To address the consistent omissions detailed in section 3.3, I have developed an open-source Java-based library—Java Simulation Infrastructure Toolkit (JSIT)—which works towards a solution for the partially-missing aspects. JSIT only provides the four section 3.3 capabilities; it is not an attempt to implement the entire reference architecture. It works with existing simulation toolkits which, as section 3.3 discussed, each implement their own partial mapping to the reference architecture. (When I use ‘toolkit’ henceforth in this section, I mean the toolkit JSIT is being used with, not JSIT itself.)

4.2 JSIT should be usable with any Java-based simulation (or one that can interoperate with Java) but, in terms of use with specific simulation frameworks, it has currently only been ‘proven’ for AnyLogic and Mason. Using it requires that the base simulation is coded in particular ways, and the degree of integration-specific ‘glue code’ required depends on the toolkit used. The idea is that JSIT helper libraries will exist for commonly-used toolkits that do most of this work for the modeler in a generic way, leveraging any useful features in the toolkit. Currently, only a helper library for AnyLogic has been written, but there is also a sample Mason-based model that shows how to integrate with JSIT in a ‘raw’ way (i.e., without use of a helper library, which effectively means implementing a simplified version of the helper library functionality as part of the model; since this does not have to be generic, it can take various short-cuts to work just for the model in question).

4.3 AnyLogic was explicitly chosen as the initial helper library focus because, compared to Mason and Repast Symphony, I expected it to have the most implementation issues (and thus the most potential influence on how the core JSIT code would need to work) because it is the most oriented towards being an integrated, visual development tool (see figure 4). My hypothesis was that the user culture focus (and commercial interests) might compromise the technical architecture, in the sense that the user-oriented features and conceptual model ‘leak into’ the user-visible parts of the technical architecture (or result in less developer focus on it) in a way which might impede the implementation (I believe this hypothesis to be correct, although I cannot be sure until the other implementations are fully complete). (I should also point out that AnyLogic-specific features also added in some aspects of the implementation.) In any case, outlining the specifics in the AnyLogic case is interesting, and gives an insight into some more general ideas (see section 6).

4.4 Figure 7 shows how a model uses JSIT. The core user-written model is some ‘root’ class which aggregates a set of model components (which may themselves aggregate other components). The root class is coded to specify itself as a JSIT-specific main model, and then uses JSIT features via the required interfaces. The helper library (or equivalent user-written code) provides the underlying link to the simulation toolkit used, but is never called directly by the core user code (i.e., the interaction with JSIT is always via a toolkit-agnostic core API).

4.5 Much more detail, together with the case study model (section 5) and sample Mason-based model, are available with full source code at https://github.com/sprossiter/JSIT. A user guide is included.

Figure 7: A UML class diagram showing the main classes and relationships when a model uses the JSIT framework, and how this splits into core JSIT code, JSIT helper library code (which might be coded by the user if no helper library exists), and user model code. Extra Java interfaces related to the EventManager class are not shown. For a background to UML, I recommend Fowler (2004).

4.6 I now explain briefly how each of the aspects in section 3.3 is implemented, with some notes on the AnyLogic helper library implementation.

Logging

4.7 This reuses a widely-used, open-source Java logging framework: Logback[27] Simulation-specific message header information (simulated date and time) are added in a generic way, and diagnostic log files are split per simulation run in multi-run batch experiments. The advanced features of Logback (compared to other Java logging frameworks) were needed to implement the per-run separation. User code just uses standard Logback mechanisms to get an appropriate Logger and log messages at differing detail levels. A per-run configuration file is used to specify what diagnostic levels are required from what classes.

4.8 The AnyLogic integration was particularly problematic because AnyLogic has some strange threading strategies, for example, single-run experiments can have models initialized in a different thread than the one time is stepped within, and certain circumstances cause models to switch execution to totally different threads. This significantly complicates the logic needed to separate log files per run, and involves AnyLogic-specific alternatives to the normal way that JSIT logging works.

Event-Driven Separation

4.9 JSIT provides a simple EventManager class and a set of interfaces that event sources and receivers need to implement. Event sources declare whether an event is a domain event or not, so the framework can also be used for non-domain-events (see section 5 for an example). The EventManager writes all domain events to a special domain events log file, which provides the model’s narrative (and reuses the logging solution). This proof-of-concept implementation is restricted to synchronous messaging, i.e., those objects subscribing to certain domain events receive them immediately on creation. Objects can subscribe to events from all instances of a source class, or just to those for a specific instance.

4.10 There are no particular AnyLogic integration issues.

Run-Reproducibility Support

4.11 When a model is run, the helper library ModelInitialiser subclass (see figure 7) automatically records environmental information and all the model parameters, using Java objects for this information which are then

http://jasss.soc.surrey.ac.uk/18/3/9.html 7 21/10/2015
serialised to an XML file (via the open source XStream library[28]) in a fairly human-readable form. Run-reproducibility information is also included at the start of the diagnostics log.

4.12 Reproducibility is significantly aided by storing model code in a version control system (VCS)—see Wilson et al. (2014). If so, environmental information is recorded on the location of this model version's code so that the exact version can be restored as needed.[29] The information also specifies if the model code had been amended from the version retrieved from the version control system.

4.13 The automated reconstitution of a model run from the information in this file is much more complicated to implement than its creation (and has not yet been done), though the object-serialised nature of the file means that it is easy to recreate the objects that it was created from. (But that just provides the model parameters and environment details; that environment ideally should still be constructed in some automated way where needed.)

4.14 AnyLogic model parameters are held in a particular way in the Java code generated for the root class. This means that model parameters can be retrieved in a generic way without requiring the user to specify or define them in any particular way. However, one key model parameter is the random seed value (if stochastic), which AnyLogic does not make accessible (largely due to restrictions in Java's Random class); it is also internally generates some extra RNG instances that affect the seed that the ‘real’ one gets (and may impact reproducibility if these internal details ever change). The code works around these issues by defining its own RNG (which the model must use). AnyLogic also does not currently allow the AnyLogic version used to be determined at run-time.

4.15 It is also not normal to separate an AnyLogic model from the experiments which run it, but this is possible by defining an ‘experiments-only model’ which has a dummy root Agent which wraps the real one. (If this separation is not done, JSIT has no way of distinguishing changes to the model code from changes to an experiment.)

Test-Oriented Stochasticity Control

4.16 A set of classes (with StochasticItem as the top-level class) provide a set of probability distributions which the model includes instances of. The implementation of these distributions (in terms of sampling them normally) is still provided by the toolkit, and the helper library Sampler class (see figure 7) makes this link. However, JSIT provides the implementation for overridden sampling currently, the only override operation supported is to collapse distributions to their mean, but a number of other useful ones are intended for the future. The user sets up an external configuration file to define any overrides required for the run (and for what distributions). There are also some other forms of convenient ‘stochastic items’ supported, such as lookup tables of distributions (useful where the distribution sampled from depends on the attributes of an individual entity, such as death rates by gender and age).

4.17 A separate advantage of this solution is to standardise the representation of probability distributions across toolkits, whilst still leveraging the per-toolkit implementations. As I said earlier, most toolkits I know do not represent them as object instances, and the sampling methods provided often have confusing differences between parameters.

4.18 In terms of AnyLogic integration, AnyLogic multi-run experiments can launch parallel runs which run in the same Java virtual machine (but in different threads). This means that there needs to be very careful concurrency-related design: distributions are typically shared by all instances of a given entity type, but there need to be separate instances per run which will be accessed from different threads (and we should assume this might be required for non-AnyLogic models as well). In the generic case, the modeler uses StochasticAccessorMDC instances (see figure 7) to manage this but, because of the threading issues with AnyLogic discussed in section 4.1, there is an alternative solution for AnyLogic.

4.19 The ‘distribution lookups’ mentioned above also leverage the AnyLogic HyperArray element, which has a useful visual interface that can be used to set up lookups to Bernoulli distributions.

### A Case Study on a Health & Social Care Model

5.1 To bring everything together, an existing multi-paradigm model of health and social care (Viana et al. 2012), using AnyLogic, was designed to use the JSIT library and, where possible within the constraints of AnyLogic, conform to the reference architecture. Since one of the goals of the reference architecture is testability (section 1), this also included setting up automated tests which compare outputs to the ‘narrative’ events log.

**Architecture**

5.2 The relevant architecture is shown in figure 8, which the reader should refer to in what follows.
5.3 The model looks at eye clinic patients suffering with age-related macular degeneration (AMD). These patients are treated at an eye clinic with eye injections which can slow the central vision loss associated with AMD, but they are contending for resources with other non-AMD eye clinic patients. The more general social care needs of the AMD sufferers is also modelled, because one broader aim is to explore how health and social care interact in an ageing population. AMD sufferers and the clinic exist in a 2-D space.

5.4 Thus, the Domain Model layer consists of AMD and non-AMD patients and an eye clinic, with AMD sufferers and the clinic composed by a root ‘core model’ object (but this is never run directly). The eye clinic holds and generates non-AMD patients itself. The AMD sufferers and the root object produce domain events, such as starting and completing an appointment at the clinic. (The root object produces events for when new AMD sufferers are created, abstracting the AMD development and clinic referral process.)

5.5 The model produces detailed outputs over simulated time for the operation of the clinic, AMD sufferers’ characteristics (e.g., sight level over time), and aggregate population statistics (e.g., numbers of AMD sufferers with different social care need levels). These are also the basis for some of the run-time visualisation. Thus, the Meta-Data Capture layer includes ‘stats gatherers’ for these aspects, and a controller for the multiple per-AMD-sufferer stats gatherers. Some of these statistics are just time series of agent characteristics at regular intervals, and so can just be sampled from the agent directly. However, most of the statistics relate to specific domain events occurring dynamically (e.g., appointments being completed) and here domain events are used: they receive notification of events they have subscribed to, and can then query the relevant component for the state they need to capture.[30] In particular, the controller is notified of new AMD sufferers, and can then dynamically create a stats gatherer for that agent. A root ‘main model’ object composes all these into a simulation which can be run as a visualisation-less version of the model.[31] (There could also be multiple variant root objects representing different combinations of meta-data capture and output.)

5.6 Modeller-coded visualisation consists of: (i) a visualisation of the 2-D space with AMD sufferers and clinic represented; (ii) a visualisation of the clinic in operation, with patients and staff moving around a layout of the space; and (iii) graphs and charts corresponding to the clinic, individual AMD patient and AMD population areas captured in the lower layer. There is also navigation around the various visualisations, including being able to ‘click-through’ AMD sufferers in the spatial visualisation to go to their statistics presentation, and on from there to view the actual agent state (provided by the built-in AnyLogic visualisation). Figure 8 shows the State & Control Presentation layer objects used to achieve this (and their relationship to lower-layer classes). Both domain and non-domain events are used here to good effect. Domain events for AMD sufferer creation are used to add their visualisation to the spatial presentation, where these visualisations graphically show certain agent characteristics (such as their stage of AMD in each eye); thus, the spatial visualisation also receives domain events relating to changes in these characteristics and updates the agent visualisation accordingly. A non-domain event is used for the population statistics visualisation to be aware when the related stats gatherer has stopped collecting (typically at the end of the model). This is needed so it can provide final updates to its charts. (They otherwise update at regular intervals, but will miss some data without this event-driven update.)

Testing

5.7 To do automated tests of the whole system, the model can be set up with simplified parameters to perform logic whose outputs can be worked out a priori ‘on paper’. To test individual components, the same thing can be done but with only those components needed composed together into the test model, with fake objects used to replace domain model components not under test where needed. (A ‘fake’ is a specific SE testing term; see Görgün et al. (2013).) Figure 9 shows the case when testing the eye clinic component (which is effectively a DES model). Note how only the components needed are composed via a single test-specific root object, and AMD sufferer objects are replaced by fakes which include only the minimal functionality needed to operate as part of the clinic appointment cycle. (Most of this logic is in the existing Patient class, so the fake logic is relatively simple.)

Figure 8: A UML class diagram showing the architecture of the AMD model, emphasising the mapping to the reference architecture. The EyeClinic actually includes a separate component which schedules clinic appointments and handles the dynamic creation of non-AMD patients, but that separation is not important for the exposition here.

Figure 9: A UML class diagram showing the architecture of the reduced model used to test the DES-based eye clinic component; compare this to figure 8.
6.6

6.5

5.13

5.12

6.4 I also focused primarily on ABM modeling (partly due to the readership of JASSS, and partly due to my own experience), though I claim the ideas are universal. There is a need to demonstrate that this is so via case studies and discussion in other areas (though note that the AMD case study here is in fact a combination of ABM, DES and SD, developed by complexity scientists and operational researchers). In particular, ideas such as logging and event-driven separation seem ill-suited to continuous-time models such as SD ones, but there is still some potential for use there. For example, an SD stock representing a market price may have domain-meaningful events when certain critical values are reached, or at intervals relating to financial reporting cycles. There is no reason why an SD model could not include 'ABM-like' ideas, such as having a rule to change form when a critical event occurs elsewhere in the system. So, in fact, the adoption of the ideas here may even encourage more cross-paradigm conceptual thinking, as well as generic SE best-practice.

6.3 The JST literature is clearly poor-proof-of-concept and there is lots more I intend to do in terms of providing more helpful libraries, 'hardening' the code (e.g., so that it scales better for large simulations) and improving documentation. The more it is used on different style models in different disciplines, the more its generality will be tested and the likelihood that useful design improvements emerge. For run-reproducibility support, the idea has recently been taken forwards for general scientific computing by the Sumatra toolkit (Davison et al. 2014), and integration with that looks promising (though it is Python-based).

6.2 The reference architecture is clearly one such architecture (see appendix A), but I believe that it captures the most important decisions; i.e., the ones with most architectural impact (and thus, referring back to Fowler’s quote, the hardest decisions to change later). I am keen to collaborate directly with toolkit developers (most of whom are also active researchers) and related initiatives such as JAMES II if we see if useful consensus can be established. It would also be useful to use this in the paper to better 'place' the emerging ideas discussed in section 1 within the landscape of toolkits and conceptual/software design abstractions. The JST AnyLogic integration also highlighted that there could be value in trying to define a form of architectural contract for simulation toolkits that would guarantee consistent architectural behaviour. As one example, there should be an expectation that a non-parallel model (i.e., one that runs single-threaded) should exist in a single well-defined thread for its full lifecycle (or at least a set of slicing threads).

6.1 This paper presents a reference architecture which embodies key SE design patterns, and helps modelers understand how to create simulations which exhibit a core set of SE best-practice properties. I have tried to show how this (together with some classificatory background) helps critically understand and evaluate simulation toolkits, and that it is possible to practically apply these ideas on real models using mainstream toolkits, aided by the JST library as part of this paper) to provide four specific generally-missing capabilities.

Conclusions

6.6 However, there are also some less obvious conceptual benefits: (i) Using domain events really encourages the modeller to think about the flow of actions in their model, what are the critical things that happen, and what agents (or model components) might care about them. This is a nice complement to thinking of the model as an ‘algorithm’. (ii) Setting up the expected outputs for automated tests (and the inputs needed to achieve the test) is time-consuming, but forcing the modeler to effectively reversion their entire conceptual design. I picked up on several design fixes by doing this, as well as a number of subtle bugs. Such testing also tends to highlight unnecessary complexity in the design, because this often makes testing trickier and the modeller is encouraged to consider whether that part of the design is really necessary.

5.12 In terms of AnyLogic’s ‘getting in the way’, most of this complexity is abstracted away into the JST framework. AnyLogic is still an open platform with object-oriented extensibility, and thus the non-JST design was reasonably straightforward. One difficulty is that all the AnyLogic help, example models, and textbooks do not encourage to treat model validation as a court case also fails to make clear how much of this is an analogy for TDD. One particular problem I think should be avoided is in introducing everything by analogy and simulation-specific terminology, ‘hiding’ the generic SE origins and thus obscuring the interdisciplinary links.

5.11 As one example, there should be an expectation that a non-parallel model (i.e., one that runs single-threaded) should exist in a single well-defined thread for its full lifecycle (or at least a set of slicing threads).

5.10 In this case, the study is somewhat biased in that I developed these ideas and the JST framework as part of creating the AnyLogic-based AMD simulation model. There is a need for other modelers to work with the ideas, and the JST library, and feed back their own experiences. There is clearly significant overlap between this task and those in the previous paragraph.

5.9 Finally, there are two broader issues which are worth slightly more extended discussion, once the first issue is resolved.

5.8 When is there so little Focus? If these kinds of SE-driven ideas are important for producing bug-free, flexible and reusable simulation code which reuses design patterns tried-and-tested across years of SE research and practice, why is there so little focus on it in the simulation literature, and how even is this focus across different disciplines?

5.7 If these kinds of SE-driven ideas are important for producing bug-free, flexible and reusable simulation code which reuses design patterns tried-and-tested across years of SE research and practice, why is there so little focus on it in the simulation literature, and how even is this focus across different disciplines?

5.6 I do not think that there is a particularly deep methodological answer, or one with historical specifics per discipline. There are the same set of reasons as discussed in the broader scientific computing best-practice literature and by related organisations (see section 1): (i) there’s always been on the science, with a feeling that these SE ideas are too ‘ heavyweight’ or too ‘commercially-focused’, especially for single-language simulations; (ii) there is a huge to normal SE training for computational scientists, and there are always issues of what to remove from the curriculum if it was added; (iii) there is a perception that SE is ‘too hard’, compounded by toolkits which try to abstract away under-the- covers detail as much as possible (Braisford 2014, §2, though see Segal (2008) for an alternative viewpoint); and (iv) the current academic system does not incentivise the production of quality software, or making it open (despite reproducibility being a cornerstone of science). All these aspects are beginning to change, and progress tends to be more advanced where the research is more universally computationally intensive (e.g. bioinformatics), and where lauded science has been shown to be flawed due to software errors (Wilson et al. 2014, p.1).

5.5 In section 1.1, I tried to introduce the theme of concepts and conceptual thinking, and discussed whether these are both somewhat guilty of this. Although there is a feeling that these SE ideas are too ‘ heavy’, or too ‘commercially-focused’, especially for single-language simulations; (ii) there is little to normal SE training for computational scientists, and there are always issues of what to remove from the curriculum if it was added; (iii) there is a perception that SE is ‘too hard’, compounded by toolkits which try to abstract away under-the- covers detail as much as possible (Braisford 2014, §2, though see Segal (2008) for an alternative viewpoint); and (iv) the current academic system does not incentivise the production of quality software, or making it open (despite reproducibility being a cornerstone of science). All these aspects are beginning to change, and progress tends to be more advanced where the research is more universally computationally intensive (e.g. bioinformatics), and where lauded science has been shown to be flawed due to software errors (Wilson et al. 2014, p.1).

5.4 In that regard, this, Graham & Rabinov (2005) state in their simulation textbook that “software tools and technologies are themselves complex and adaptive, and different technologies are best for different [models]. We cannot make this a software engineering book, and if we did it would likely be out of date by the time you read it.” With sympathy with this (and their general outline of important software engineering ideas is good), this paper contends that, whilst technologies may rapidly change, good design patterns tend not to and can be usefully taught.

5.3 I believe that it captures the most important decisions; i.e., the ones with most architectural impact (and thus, referring back to Fowler’s quote, the hardest decisions to change later). I am keen to collaborate directly with toolkit developers (most of whom are also active researchers) and related initiatives such as JAMES II if we see if useful consensus can be established. It would also be useful to use this in the paper to better ‘place’ the emerging ideas discussed in section 1 within the landscape of toolkits and conceptual/software design abstractions. The JST AnyLogic integration also highlighted that there could be value in trying to define a form of architectural contract for simulation toolkits that would guarantee consistent architectural behaviour. As one example, there should be an expectation that a non-parallel model (i.e., one that runs single-threaded) should exist in a single well-defined thread for its full lifecycle (or at least a set of slicing threads).

5.2 In terms of the case study, I am somewhat biased in that I developed these ideas and the JST framework as part of creating the AnyLogic-based AMD simulation model. There is a need for other modelers to work with the ideas, and the JST library, and feed back their own experiences. There is clearly significant overlap between this task and those in the previous paragraph.

5.1 Finally, there are two broader issues which are worth slightly more extended discussion, where the first issue is resolved.

5.0 There is no reason why an SD model could not include ‘ABM-like’ ideas, such as having a rule to change form when a critical event occurs elsewhere in the system. So, in fact, the adoption of the ideas here may even encourage more cross-paradigm conceptual thinking, as well as generic SE best-practice.

Acknowledgements

The author wishes to thank Jason Noble for useful discussion and support. Joe Viana was also very patient in allowing the JST extensions to be used in the case study model. This work was conducted under the EPSRC-funded Care Life Cycle (CLC) project, which is EPSRC grant EP/H021689/1.

Notes

1See http://www.software.ac.uk.


4There are also a large number of references to empirical studies in the best-practice papers and books mentioned.

5I use the term ‘toolkit’ generically to refer to all the existing platforms for simulation which, as well as ‘platforms’ and ‘toolkits’, can also be referred to as software libraries or frameworks. There are some technical distinctions between these terms, but they are not relevant for the purpose of this paper.

6The foundational theory of modeling and simulation in the DEVS literature (Ziegler et al. 2000)—which extends to ABM (Müller 2009)—sets up some very useful frames and terminology to talk about these issues more formally for simulation (which I unfortunately do not have space to touch on here), but does not directly address them.

7This would also help practitioners understand the contribution of the emerging ideas just mentioned.

8There is a reasonable amount of focus across these references on the development process, but that is not the same thing. There is also a growing literature on model comparison and reproduction (Rouchier et al. 2008), but the ‘reproducibility’ there (if model results separate from implementation details of the same model or potentially equivalent models) is in a different sense to the reproducibility here (of runs for the same model implementation).

http://jasss.soc.surrey.ac.uk/18/3/9.html

21/10/2015
Appendix A: Alternative Reference Architectures

A.1 Since any reference architecture is a subjective assessment of how to partition functionality for an application domain, it is worthwhile briefly comparing the reference architecture given here with some partial or full alternatives.

Model Instance as a Domain Model

A.2 Conceptually, it may feel that the domain model separation in figure 1 is not quite right, because the particular domain model for the experiment is also defined by its inputs (part of the Experiments Definition layer). In ABM, there is also often a separation between the domain model components (agents) and their composition into a model; for example, Repast Symphony (North et al. 2013) uses "contexts" for this purpose. Thus, figure 10 might seem better in this regard.

![Diagram](http://jasss.soc.surrey.ac.uk/18/3/9.html)
Appendix B: Example JSIT Outputs from the Case Study

B.1 To make things more concrete, below are some extracts from the case study events log, diagnostic log, and run-reproducibility settings file. Figure 12 shows the events log. Notice the simulated day and time in the header for each event, and how it looks as a ‘narrative’ of the events. The inclusion of details such as the length of stay at the end of each appointment allows the log to be used for automated testing. Figure 13 shows the diagnostic log. Note how this is a more ‘programmer-friendly’ message header (with thread ID, class producing the message and detail level). Typically, INFO is the default message level and is used for messages generally of interest to the user in a generic sense. However, the object-oriented extensibility of toolkits like AnyLogic sits a bit awkwardly here because models are not clean object-based-style compositions of pre-built elements (such as the microsimulation models on the MicMac project (Zrin et al. 2013) using a custom-designed JAMES II simulator McCord). Secondly, other than the registry, they do not explicitly include any cross-cutting aspects in their architecture; hence they seem less concerned with ‘pragmatic’ SE best-practice than I am. (Basicall, the imperative aspects of their architecture differ in focus and granularity to mine.)

Figure 12: Sample events log output (extracts) from the case study model using the JSIT library.

```
DAY 1 05:10 AMD Sufferer #1 had an AMD.Initial appointment booked for day 2
DAY 1 05:10 Created referred AMD Sufferer #1: FEMALE aged 71
DAY 1 09:00 Non-AMD Patient #4 started appointment type OTHER without FFA or OCT
DAY 1 09:05 Non-AMD Patient #5 started appointment type OTHER without FFA or OCT
DAY 1 09:46 Non-AMD Patient #4 completed their OTHER appointment: length of stay 46, total wait time 0
DAY 1 09:57 Non-AMD Patient #5 started appointment type OTHER without FFA or OCT
DAY 2 09:00 AMD Sufferer #1 started appointment type AMD.OTHER with FFA
DAY 3 09:42 AMD Sufferer #1 had AMD injection(s) in affected eye(s)
DAY 3 10:53 AMD Sufferer #1 completed their AMD.INJECTION appointment: length of stay 53, total wait time 0
DAY 3 10:20 Non-AMD Patient #13 started appointment type OTHER with FFA
DAY 3 13:04 AMD Sufferer #1 completed their OTHER appointment: length of stay 71, total wait time 0
DAY 3 10:47 AMD Sufferer #17 started appointment type OTHER with FFA
DAY 3 11:03 Non-AMD Patient #19 started appointment type OTHER with FFA or OCT
DAY 3 11:08 Non-AMD Patient #14 started appointment type OTHER without FFA or OCT
DAY 3 12:15 Non-AMD Patient #19 completed their OTHER appointment: length of stay 71, total wait time 0
DAY 35 09:58 AMD Sufferer #34 has died aged 96
DAY 7 05:27 AMD Sufferer #3 changed AMD level from EARLY to INTERMEDIATE
DAY 9 01:00 AMD Sufferer #2 changed care need from CRITICAL to MODERATE
DAY 9 01:00 AMD Sufferer #3 changed care need from MODERATE to CRITICAL
DAY 9 01:00 AMD Sufferer #4 changed care need from NOHE to MODERATE
DAY 9 01:00 AMD Sufferer #4 changed care provision from NOHE to INFORMAL ONLY
```

Figure 13: Sample diagnostics log output (extracts) from the case study model using the JSIT library.

```
<environmentSettings>
  <modelName>AMD Whole-System Health & Social Care</modelName>
  <modelVersion>0.2</modelVersion>
  <modelVCS>Svn</modelVCS>
  <modelVCSVersionSource>https://svn.soton.ac.uk/CLC_Project/Simulations/AMD_HealthSocialCare/Trunk/AMD_Model/Coder/modelVCSVersionSource</modelVCSVersionSource>
  <runtimeCodeHash>c1f4e9e12272f3b56bca241e473</runtimeCodeHash>
  <modificationStatus>NO</modificationStatus>
  <javaVersion>1.8.0.05</javaVersion>
  <javaVM>Java HotSpot(TM) 64-Bit Server VM 25.5-b02 (Oracle Corporation)</javaVM>
</environmentSettings>
```
References


