# Story-thinking, computational-thinking, programming and software engineering

**Austen Rainer**
School of Electronics,
Electrical Engineering
& Computer Science
Queen's University Belfast
a.rainer@qub.ac.uk

**Catherine Menon**
School of Physics,
Engineering
& Computer Science
University of Hertfordshire
c.menon@herts.ac.uk

## Abstract

Working with stories and working with computations require different modes of thought. We call the first mode "story-thinking" and the second "computational-thinking". The aim of this paper is to explore the nature of these two modes of thinking, and to do so in relation to programming, including software engineering as programming-in-the-large. We use two stories as illustrative examples: a famous six word story and the Byzantine Generals Problem. With these two stories, we explore one fundamental problem, i.e., the problem of "neglectful representations". We briefly suggest ways in which this problem might be tackled, and briefly summarise our ongoing investigations.

## 1. Introduction

The term "story" is widely used in software engineering, e.g., the "user story" (Lucassen, Dalpiaz, Van Der Werf, & Brinkkemper, 2015) and the "job story" (Lucassen et al., 2018). Related concepts are also used, e.g., the scenario (Carroll, 2003). But *story* – in the fullest sense of that word – and algorithm are very different things. We use the term "story-thinking" to refer to the ways in which we conceptualise and engage with stories, e.g., how we create them, how we formally analyse and evaluate them, and how and why we appreciate them. We use the term "computational-thinking" to refer to the ways in which we think about computations, e.g., how we create algorithms, how we formally analyse and evaluate them, and how we simulate their execution on real or nominal machines. The contrasts between story-thinking and computational-thinking stimulate a range of questions, e.g., what are the ways in which story and algorithm might relate? how might story – its writing, telling and re-telling – complement computational thinking? and what do we we gain, and what do we lose, with these contrasting ways of thinking?

The aim of this paper is to explore the nature of these two modes of thinking, and to do so in relation to programming, including software engineering as programming-in-the-large. The paper contributes the following:

1. A conceptualisation of some of the issues, through the application of these two modes of thinking to two illustrative examples.
2. The identification of a fundamental problem: the problem of "neglectful representations".
3. Brief proposals for how these problems might be tackled.

It is helpful, at the outset, to briefly review definitions of computational thinking. Aho (2012) defines computational thinking as, "...the thought processes involved in *formulating problems* so their *solutions* can be *represented as computational steps and algorithms*." (emphasis added). Denning (2009) recognises the *representation* as more fundamental than the algorithm, since the representation needs to be computable (Erwig, 2017) or, on other words, manipulatable by or through computation. Heineman, Pollice, and Selkow (2008) write that, "Designing efficient algorithms often starts by selecting *the proper data structures* in which to represent the problem to be solved." (emphasis added).

Priami (2007) highlights a particular feature of the representation: "...the basic feature of computational thinking is abstraction [representation] of reality in such a way that the *neglected details* in the model make it executable by a machine." (emphasis added). This concept of negative selection – i.e., of what is

*removed* from the model *in order to* allow or support or enable computation – is particularly interesting because the focus with abstraction tends to be on what is retained, i.e., on *retaining* only those features of the thing to be modelled that are *essential* (Starfield, Smith, & Bleloch, 1994).

The remainder of the paper is organised as follows. We start our exploration with an illustrative example, a six word story, in Section 2. In Section 3, we model the example computationally. In Section 4 we introduce and discuss a second illustrative problem, the Byzantine Generals Problem. In Section 5, we then consider computational thinking in the context of software engineering, as programming-in-the-large. In Section 6, we then summarise the problems and propose ways forward. Finally, with Section 7, we briefly conclude.

## 2. First illustrative example: the sale of shoes

Consider the following short story (which has been intentionally modified slightly from a well-known story in the literary world):

> "For sale. Baby's shoes. Never worn."

What is your reaction to this story? In his book, *Once upon an if*, Worley (2014) describes his wife's first reaction to this story. "Oh, no!" he says she responded, an indication that she interpreted the story as one of sadness or tragedy, presumably relating to the life of a baby. As the reader, you might, similar to Worley's wife, interpret the story as a tragedy. Or you might, as Worley says his wife's friend did, interpret the story as about someone who buys things for others but those things are not wanted. And, of course, you might have other reactions to the story. In a recent online presentation of this story to software engineering academics, the audience was asked for their reactions. Responses included, "love, compassion", "sadness", "cute", and "nostalgic – story made me reflect on that episode of my life". Conversely, in a recent teaching activity with undergraduate students, the students were asked to brainstorm possible explanations for the events described in the six words. One suggested explanation was that the six words are simply an advert, albeit a little oddly phrased, for the sale of new shoes. One might imagine this advert in the window of a shop, with the message now "de-particularised." There is no longer a protagonist (unless you count the shop keeper) nor a plot. There are just shoes for sale.

Furthermore, the reference to a *baby* means the phrase "never worn" carries different connotations compared to, for example, an adult having "never worn" the shoes. The connotations of a baby's shoes never being worn are tragic because babies are seen as more susceptible to, and "fitting" (in story-terms) for, tragedy. An adult's "never worn" shoes may imply the shoes have been worn at least once, for example to try them on, but then not worn again, e.g., perhaps they didn't fit.

Whatever your interpretation of the story, and your reaction to it, the story-teller (the story is based on a story allegedly written by Ernest Hemingway, though this is disputed) has provoked an experience and done so very efficiently. In just six words, the story-teller creates characters (e.g., a baby, and possibly a parent), a plot (e.g., perhaps someone has lost a baby, or not been able to have a baby) and therefore an unfolding of events over time, one or more goals and a struggle (e.g., the goal of having a baby, with the struggle of not having a baby, or of surviving the loss of a baby), an outcome (e.g., the goal was not attained) and an emotional experience for the reader (e.g., sadness).

The entire story is shorter in length than Cohn's (2004) well-accepted *template* for a *single* user story in requirements engineering: `As a <type of user> , I want <goal>, [so that <some reason>]`. One reason we chose this story as an illustrative example is because it is concise and therefore easy to present completely in an academic publication with restrictions on page length. But we also chose this story because it is efficient: an extraordinary amount of information and emotion is evoked in only a few words. This efficiency contrasts with the user story. The contrast – between the six-word story and the template for a single user-story – suggests fundamental differences in the way that stories model the world and the way that typical software engineering and programming constructs model the world; and also suggests fundamental differences in the ways that story–thinking and computational-thinking

require us, or encourage us, to *attend* to the world.

## 3. Representing the story computationally

An alternative way to think about the six-word story is computationally. And it seems that as soon as we start to attend to the six-word story computationally it is no longer a *story* but instead becomes a text. To explore this point we present and consider two forms of text associated with computational thinking: user stories and software designs.

### 3.1. Representing the story as user stories

Table 1 presents examples of *possible* user stories, first re-stating Cohn's (2004) template for user stories. These user stories can only be speculative because they depend on how one interprets the six words.

| ID | Example |
|---|---|
| N/A | As a <type of user> , I want <goal>, [so that <some reason>] |
| 1 | As a user, I want to be able to sell items, so that I can make some money. |
| 2 | As the purchaser of an unwanted item of clothing, I want to sell the item, so that I can recover my costs. |
| 3 | As a grieving parent, I want to sell my baby's shoes, so that I can reduce my financial losses. |

*Table 1 – Example user stories for the thought experiment*

The examples in Table 1 illustrate some of the tensions, strengths and weaknesses of computational-thinking and story-thinking, such as:

1. The *story* tells us very little explicitly about the actual protagonist, other than the protagonist wants to sell a pair of baby's shoes. We are left to infer the characteristics of the protagonist, including the basis of their goal. In terms of story-thinking, this is an effective rhetorical device. In terms of computational-thinking, this is problematic. There is a tension between story-thinking's use of evocative connotation and computational-thinking's standards of specification and denotation.

2. There is also a tension between story-thinking's particularity and computational-thinking's abstraction. The users, or personas, defined in User Stories 2 and 3 can be abstracted to the user defined in User Story 1. Conversely, whilst the emotional aspects of the user in User Story 3 (e.g., sadness, nostalgia, not wanting something) might be *representable* in a software system, it is not clear what gain there is *for the software system* with such a representation. Or in other words, it is not clear how representing such states in the system helps the user with what they want to *do*.

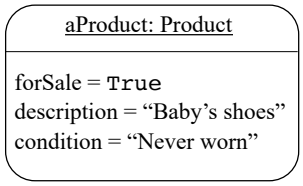
   This distinction between particularity and generality can be further illustrated by comparing the version of the story presented at the beginning of Section 2 with the well-known, original story. In the original version, the story reads, "For sale. Baby shoes. Never worn." Adding one apostrophe together with one letter, the letter *s*, helps to particularise the story, e.g., by adding a new character, a specific baby. There is no longer the abstract category, baby shoes; nor is there a collection of babies' shoes. Instead there is "[a] baby's shoes". There is another, perhaps more subtle, particularising effect too: the word "Baby" can often be used as a name, e.g., "Have you fed Baby yet?" or, "Don't wake Baby."

3. In the act of preparing User Stories to model the story, we begin to reshape our conception of the story. We specify, and possibly also abstract, and by doing so we change the story, limit it and reduce its effect as a story.

### 3.2. Representing the story as a software design

A second way to think computationally about the story is in terms of software designs. Figure 1 presents a simple UML object diagram (Figure 1a) and a database table (Figure 1b) for the sale of a product, in this case a pair of baby's shoes. We can of course think with and about these models, but notice how the *kind* of thinking we do with these models is different to the kind of thinking we do with the *story*.

With the object diagram we can, for example, infer that the object is an instance of the class `Product`, and we can see the `forSale` attribute is a boolean variable. We can also see that no private or public methods are stated for this object, at least in the diagram, and we might examine the class `Product` for such methods. The database table shows that we can easily begin to design a database for persistent storage of information taken from the story. Being outputs of our thinking, these models provide insights into the nature of our computational-thinking.




*(a) UML object diagram*

| ID | Type | Subtype | Description | Condition | Size |
|---|---|---|---|---|---|
| AF1G20 | Shoes | Baby | Baby's shoes | Never worn | Small |

*(b) Database table*

*Figure 1 – UML object diagram and database table for the sale of a product*

And we can, of course, revise the models. We might change the `condition` attribute to become an enumerated variable, perhaps using the value of `1` to represent the condition of "Never worn." An enumerated variable would help us implement other features, e.g., with an SQL database, it becomes easier to select products that are never worn, e.g., `SELECT * WHERE Condition == 1`. An enumerated variable also improves the efficiency of computational processes, e.g., an `if` test of a numeric variable requires less computational resource than an equivalent test of a string value.

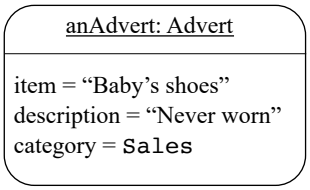



*Figure 2 – A UML object diagram for an item to advertise*

The object model presented in Figure 1 is not the only model that might be constructed. Figure 2 suggests a different object model. Comparing Figure 1 and Figure 2, the same *text* is present but the way we have *structured* the text is different. Our interpretation of the six words leads to different computational representations and those representations then support different thinking, e.g., we might enumerate the `category` attribute and, with the appropriate database design, select only the sale adverts: the SQL statement, `SELECT * WHERE category == 1`, now means something different. Though the model has changed, the fundamental nature of the model hasn't, and the kind of thinking – computational thinking – hasn't changed either.

The UML examples also help to illustrate that one feature removed from the story is not the literal words but the *order* to the words. Removing the order "dismantles" the story. Haven (2007) presents an example to illustrate how simply varying the placement of a word effects the meaning of a sentence. Compare the following examples, taken from Haven's book, *Story Proof* (Haven, 2007):

John will marry Elise.
*Even* John will marry Elise.
John will *even* marry Elise.
John will marry *even* Elise.

Notice not just that the meaning of the sentence changes but also, if we dwell on each sentence, we might start to wonder about the context and the motivation for John, e.g., what might be going on for John to marry *even* Elise?

As well as "dismantling" the story, removing the order of the words so as to make a computable representation also 'de-means' – i.e., reduces the very meaning of – the story for a human whilst, conversely, formalising the representation to enable computation. At the same time, details are *added* to the computational model, e.g., data types, methods, class-object structures.

## 4. Second illustrative example: the Byzantine Generals Problem

Removing information in order to enable computation can be illustrated through another story, the Byzantine Generals Problem (BGP). There are several versions of the BGP reported in the computing literature. We take what is perhaps the most commonly-cited version, published within the safety engineering community by Lamport, Shostak, and Pease (1982).

> . . . several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, **they must decide upon a common plan of action**. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that the following two conditions are met:
> #1: All loyal generals **decide on the same plan of action** [. . . ]
> #2: A small number of traitors cannot cause the loyal generals to adopt a bad plan.
> (emboldened emphasises added)

The BGP was formulated by computer scientists to illustrate a particular *computational* problem, i.e., ensuring reliable communication in the presence of faulty components. In a previous publication, we (Menon & Rainer, 2021) critically evaluated the BGP story, concluding that the story 'fails' *as a story*. For example, the story lacks any *humanly-meaningful* objective. Generals are expected to agree on a common plan of action, but it does not matter on what they agree. They can attack or retreat or, in principle, do anything else they agree on. Their objective is arbitrary. This is odd for a story because usually in a story a character would have some motive for their objective.

It is conceivable that, for some stories, there are characters who just want everyone to agree, but don't care what they agree on. A good example is a waiter trying to take an order for a shared pizza from a family of four. But in this example, there are other characters – those in the family – who do want particular outcomes. And also, generals arguing over strategy don't fit this kind of story.

To make a more effective story for the reader (a *reader*, not a software engineer or a programmer or, more generally, a computational-thinker), what is missing from the sentence is a final clause, i.e., decide to do what? Furthermore, since the loyal generals' objective is arbitrary, the traitors' objectives are also arbitrary: the traitors are only concerned with preventing an arbitrary legitimate agreement. But note too that not only must the traitors' objectives be arbitrary relative to the loyal generals, *each* of the traitor's objectives must be arbitrary relative to all other traitors. In this context, each traitor might be better understood as an agent of chaos.

The BGP therefore *risks* misrepresenting the computational problem through the way it presents a story in terms of human agents. To prevent this risk, to "square the circle" between story-thinking and computational-thinking, the human agents in the BGP are given odd (for a human) intention. Safety engineers possess the technical knowledge needed to interpret the story "correctly" *for the computational problem* being explored. For safety engineers, as computational-thinkers, the arbitrariness of the objectives is not just acceptable but essential. This is because the arbitrariness of the objectives allows for an algorithmic solution that has wide applicability: the algorithmic solution would apply for situations where generals agree to attack and for situations where generals agree to retreat and for situations where generals (arbitrarily) agree to do something else. The BGP abstracts the problem so as to provide a generalised solution (compare with the abstraction of User Story 2 and 3 to Use Story 1, in Table 1). And in abstracting the problem it must necessarily remove a humanly-meaningful quality of the story, i.e., meaningful human intention.

Overall then, the BGP acts as a kind of mirror example to the six-word story. To go from the six-word story to the computational representation, we remove something essential. To go from the computational problem to a BGP *story* we would need to add something essential, i.e., consistent characterisation and motivation, and the failure to do so contributes to its failure as a story. In contrasting ways, both neglect. The BGP is not meaning*less* – we can still understand the story – but it is not meaning*ful*, in human terms, as a "good" story.

## 5. Software engineering: programming-in-the-large

For conciseness, we use Johnson and Ekstedt's (2016) Tarpit Theory of Software Engineering as our reference for discussing the nature of software engineering. As part of the summary of their theory, Johnson and Ekstedt write, "The goal of software development is to create programs that, when executed by a computer, result in behavior that is of utility to some stakeholder."

Drawing on the discussion of computational thinking, we might say that software is *useful* – of utility – to a stakeholder when the software behaves in a way that solves a *representation* of a problem experienced by that stakeholder. Solving a representation of a problem is not the same as solving the problem. And inferring from Priami's (2007) assertion, in Section 1, as well as from our discussion of the BGP, some problems must be *neglected* in order to make the resulting model computable.

Johnson and Ekstedt (2016) also write that, "Much of software engineering concerns translations; design specifications are translated into source code…, source code is translated into machine code…, etc. In fact, the whole process of software engineering can be considered as a series of translations…(Johnson and Ekstedt (2016), p. 187). They define *language* as "A set of specifications", *translation* as, "An activity that preserves the semantic equivalence between source and target languages [specifications]," and *semantic equivalence* as, "Two specifications are semantically equivalent if, when translated to a common language, they are syntactically identical." (Notice how *semantic equivalence* is being defined in terms of *syntactic identity*.)

We have shown, with the six-word story and the BGP story, that it is not necessarily feasible to translate the story from the source language to a target language *and* maintain semantic equivalence. More than that, it is unreasonable to *expect* a translation, in the way that Johnson and Ekstedt define it, from the English language of the six-word story to, as examples, the User Story, the object diagram or the database table. But that is the point. The language used with story-thinking, and therefore the representations used for story-thinking, are not semantically equivalent to the language and representations used for computational-thinking.

Johnson and Ekstedt (2016) seem to recognise this problem of non-equivalence, when they write:

> This leads us to *the major challenge of software engineering*, that programs – these formal, static, syntactic `compositions` – are very different from the oftentimes **fluid and intangible stakeholder experiences** they are intended to **evoke**. A significant feat of **imagination** is thus required on the part of the `developers` to predict the effects of a given `syntactic` modification to the program code on the end-users's `experiences`. An even greater challenge, which we propose as the *core task of software* `endeavors`, is **to determine which `syntactic` manipulation will cause a specified `stakeholder experience`**, and then to perform the appropriate informed manipulations, i.e. to make `design decisions`.
> (In the original text, the `san serif` typeface denotes formal constructs of the theory, whilst the *italicised* text are the original authors' emphasise. **Emboldened** emphasis are ours. We recognise these variations in typeface and font make the text more confusing, visually, but retain them to be faithful to the original.)

Clearly, Johnson and Ekstedt (2016) are aware of the "gap" between stakeholders' experiences and executable programs, and of the challenge of bridging this gap.

## 6. A summary of the problems and suggested ways forward

Drawing on the preceding discussion, we summarise one specific problem, briefly suggest ways in which this problems might be tackled, and briefly describe ongoing investigations we are conducting in this area.

### 6.1. The problem of neglectful representations

As Priami (2007) observes, computational thinking neglects. More than that, computational thinking must neglect. It does this in order to arrive at a representation that is computable. These "neglectful representations" inevitably "de-mean", i.e., reduce humanly-meaningful qualities. Neglectful representations have implications for the impact of computational-thinking, programming and software engineering on society, the economy and the environment. For example, economic impact is much easier to align with computational-thinking because economic thinking appears to be a way of thinking similar in kind to computational-thinking. But humanly-meaningful qualities of society, as well as the qualities of the environment, are much harder to represent, a point that is increasingly recognised, e.g., with work on values in computing (Ferrario et al., 2017), kind computing (Alrimawi & Nuseibeh, 2022), compassionate computing (Pomputius, 2020) and responsible software engineering (Schieferdecker, 2020).

This problem of neglectful representation and, by comparison, the complementary use of story as a possible solution, appear to be implicitly recognised by others in software engineering. Strøm (2006, 2007) investigates the role and value of stories that include emotions and conflicts in software engineering. Bailin introduced design stories (Bailin, 2003) and also discussed how features need stories to convey the "missing semantics" that address "fine-grained questions of context, interface, function, performance, and rationale" (Bailin, 2009). In an unpublished paper, Stubblefield et al. (2002) observe that, "In our experience, most software development projects do not fail for technical reasons. They fail because they do not engage users at the fundamental level of value, meaning and practice. They solve the wrong problem, or fail to support deeper patterns of work and social interaction." (Stubblefield & Rogers, 2002). Finally, Yilmaz et al. (2016) report a preliminary study to demonstrate the benefits of story-driven software development: "... it is important to capture and store the excessively valuable tacit knowledge using a rich story-based approach."

### 6.2. Suggested ways forward

We suggest two directions for future research:

1. Developing methodology that connects stories with algorithms. In a previous paper, Rainer (2017), drawing on prior work in law and legal reasoning, proposed a methodology for identifying arguments and stories from blog articles, extracting them, and then graphically combining them. This methodology might be extended or, alternatively, might provide an example for a potential methodology to integrate representations used for story-thinking with representations used for computational-thinking.

2. Changing software practice. With the agile methodology, software practitioners return to the user story, e.g., its acceptance criteria, to evaluate whether a user story has been delivered. We suggest that practitioners might return to the *story*, and not just the user story. As one example, a software engineering (SE) team developing a software system for managing information about children in publicly–funded care homes might analyse Sissay's memoir, *My Life is Why* (Sissay, 2019), to gain insights into the experiences of children in care, and therefore into the meaningful user stories for such a persona or stakeholder.

### 6.3. Ongoing investigations

We have four ongoing investigations in this area:

1. A closer study of the Byzantine General Problem, in which the Problem is represented as user stories and one then attempts to implement those user stories in executable code.

2. A study of Sissay's memoir, *My Life is Why* (Sissay, 2019), in which, again, the memoir is represented as user stories, as well as other requirements, and one then attempts to implement the user stories into database designs and then executable code.

3. A study of how writers think about theirs, and others', short stories, and how formal techniques from software engineering might support their thinking.

4. A study of how GPT-3 responds to prompts about stories such as the six-word story.

## 7. Conclusion

In this paper we explored two modes of thinking: story-thinking and computational-thinking. Using two examples – a six-word story and the Byzantine Generals Problem – we show how story-thinking and computational-thinking attend to these stories in different ways. We considered how software engineering, as programming-in-the-large, recognises, at least implicitly, these different modes of thought, as well as the challenges of integrating these modes. We identified the problem of neglectful representations, briefly suggested ways in which these problems might be tackled, and briefly summarised our ongoing investigations.

## 8. Acknowledgements

We thank the reviewers and the conference attendees for their constructive and supportive feedback. We also thank participants of the online presentation for permission to quote them, and the students of the brainstorming session.

## 9. References

Aho, A. V. (2012). Computation and computational thinking. *The Computer Journal*, *55*(7), 832–835.

Alrimawi, F., & Nuseibeh, B. (2022). Kind computing.

Bailin, S. C. (2003). Diagrams and design stories. *Machine Graphics and Vision*, *12*(1), 17–38.

Bailin, S. C. (2009). Features need stories. In *International conference on software reuse* (pp. 51–64).

Carroll, J. M. (2003). *Making use: scenario-based design of human-computer interactions*. MIT press.

Cohn, M. (2004). *User stories applied: For agile software development*. Addison-Wesley Professional.

Denning, P. J. (2009). The profession of it beyond computational thinking. *Communications of the ACM*, *52*(6), 28–30.

Erwig, M. (2017). *Once upon an algorithm: how stories explain computing*. MIT Press.

Ferrario, M. A., Simm, W., Whittle, J., Frauenberger, C., Fitzpatrick, G., & Purgathofer, P. (2017). Values in computing. In *Proceedings of the 2017 chi conference extended abstracts on human factors in computing systems* (pp. 660–667).

Haven, K. (2007). *Story proof: The science behind the startling power of story*. Greenwood Publishing Group.

Heineman, G. T., Pollice, G., & Selkow, S. (2008). *Algorithms in a nutshell*. O'Reilly Media.

Johnson, P., & Ekstedt, M. (2016). The tarpit–a general theory of software engineering. *Information and Software Technology*, *70*, 181–203.

Lamport, L., Shostak, R., & Pease, M. (1982). The byzantine generals problem. *ACM Transactions on Progamming Languages and Systems*, *4*(3), 382-401.

Lucassen, G., Dalpiaz, F., Van Der Werf, J. M. E., & Brinkkemper, S. (2015). Forging high-quality user stories: towards a discipline for agile requirements. In *2015 ieee 23rd international requirements engineering conference (re)* (pp. 126–135).

Lucassen, G., van de Keuken, M., Dalpiaz, F., Brinkkemper, S., Sloof, G. W., & Schlingmann, J. (2018). Jobs-to-be-done oriented requirements engineering: a method for defining job stories. In *International working conference on requirements engineering: Foundation for software quality* (pp. 227–243).

Menon, C., & Rainer, A. (2021). Stories and narratives in safety engineering. In *Proceedings of the 30th safety critical systems symposium, volume: Scsc-170.* Retrieved from `https://scsc.uk/scsc-170`

Pomputius, A. (2020). Compassionate computing in the time of covid-19: Interview with laurie n. taylor. *Medical Reference Services Quarterly*, *39*(4), 399–405.

Priami, C. (2007). Computational thinking in biology. In *Transactions on computational systems biology viii* (pp. 63–76). Springer.

Rainer, A. (2017). Using argumentation theory to analyse software practitioners' defeasible evidence, inference and belief. *Information and Software Technology*, *87*, 62–80.

Schieferdecker, I. (2020). Responsible software engineering. In *The future of software quality assurance* (pp. 137–146). Springer, Cham.

Sissay, L. (2019). *My name is why*. Canongate Books.

Starfield, A. M., Smith, K. A., & Bleloch, A. L. (1994). *How to model it: Problem solving for the computer age*. Interaction Book Company.

Strøm, G. (2006). The reader creates a personal meaning: A comparative study of scenarios and human-centred stories. In *People and computers xix—the bigger picture* (pp. 53–68). Springer.

Strøm, G. (2007). Stories with emotions and conflicts drive development of better interactions in industrial software projects. In *Proceedings of the 19th australasian conference on computer-human interaction: Entertaining user interfaces* (pp. 115–121).

Stubblefield, W. A., & Rogers, K. S. (2002). *The micro traveler and the hero's journey.* Retrieved from `https://wmstubblefield.com/wp-content/uploads/2019/06/narrativePaper.pdf`

Worley, P. (2014). *Once up an if: the storythinking handbook*. Bloomsbury.

Yilmaz, M., Atasoy, B., O'Connor, R. V., Martens, J.-B., & Clarke, P. (2016). Software developer's journey. In *European conference on software process improvement* (pp. 203–211).