Story and Software

Konrad Mosoczy

***Abstract*—Background:** Stories have been a part of human culture and society at large for centuries. Stories have also found their place in software engineering, e.g. as scenarios and user stories. Yet the relationship between how we reason with stories and how we reason algorithmically has not yet been properly investigated. **Objective:** To better understand the relationship between story-thinking and computational-thinking. **Method:** Investigated several stories found in the literature, analysing them in various ways, e.g., writing user stories, drawing up design diagrams, and ultimately implementing them in code; and developed a suite of tools; and illustrated their use through applying them to the previously investigated stories. **Findings:** Stories can be defined as being made up of a series of elements; software can also be broken down into its own set of 'essential' elements; there exist similarities between the elements of stories and the elements of software and, in fact, we are able to produce a mapping between story and software in terms of these elements; the concept of 'plot' plays a central role in this mapping. We also discovered several constraints and limitations, however, to the type of stories that can be worked with (such as complexity of the story). **Conclusion:** Working with stories is challenging. A story must undergo several transformations before being usable in a software context. Initially, computational-thinking and story-thinking seemed incompatible, however, the idea of story and software being broken down into somewhat-related elements allowed for the development of a framework to bridge the gap (to some extent) between how we reason with stories and how we reason with software.

***Index Terms*—**Stories, software, story-thinking, computational-thinking, abstractions

I. Introduction

THE earliest examples of written literature are commonly believed to have originated in ancient Mesopotamia. The Sumerian civilization first developed writing around 3400 B.C., when they began making markings on clay tablets in a script known as cuneiform [1]. The Sumerians produced a vast and highly developed literature, largely poetic in character, consisting of epics and myths, hymns and lamentations, proverbs and 'words of wisdom' [2].

Stories have, of course, existed long before the advent of writing - just consider the countless cave paintings left by our ancestors - not to mention the sharing of stories orally. From the epic of Gilgamesh to ancient Chinese poetry to the narratives we share today, stories have always had a place in human history. Aristotle, being one of the first people to consider the philosophical aspect of stories, notably said 'Poetry has sprung from two causes, each of them lying deep in our nature' (the first of these is the instinct of imitation, and the second is the instinct of harmony and rhythm) [3]. It is not too big of a stretch, then, to make the same statement about stories in general. Thus, if poetry is a part of human nature, then, by extension, so is narrative as a whole.

Konrad Mosoczy is with the School of Electronics, Electrical Engineering and Computer Science, Queen's University Belfast, UK, e-mail: kmosoczy01@qub.ac.uk

It should likely be no surprise, then, that stories have found themselves a home in today's modern world amongst Software Engineers; we need not look far for examples in this regard. How often have you heard of Matryoshka Dolls being used to explain recursion, or the real-world concept of a checkout line being used to explain queues [4]? It is important to note that the use of stories in SE need not be limited to education. Consider, for instance, the *Travelling Salesman Problem* [5] or the *Tower of Hanoi* [6]. These stories have become so well-known to, both, layman and engineer alike that they overshadow the problem they were designed to abstract away. These examples are not one-off events - the use of narrative seems to pervade many aspects of SE, even down to the language that is used. Consider how classes are said to 'know' things, or how 'packets' are sent across the internet. Metaphors and analogies such as these, while not exactly 'stories', are not so different to something like the aforementioned problems. Consider also the fact that, oftentimes, an original problem or concept for a piece of software is initially presented as a *story*, as is the case with the aforementioned famous problems. Think also about how 'user stories' and 'scenarios' are used in developing code. These are two immensely powerful and important 'tools' in requirements engineering that help software engineers develop better software. As stated by Leffingwell and Behrens, 'an argument can be made that the user story is the most important artifact in agile development, because it is the container that primarily carries the value stream to the user' [7]. Also, consider what is said by Carroll et al. regarding scenario-based software design - 'their [scenarios] vivid depiction of human activity promotes focused reflection on the usefulness and usability of an envisioned design intervention. This helps to ensure an early and technically detailed consideration of use, and the context of use, and mitigates the temptations and distortions of technology-driven development' [8]. In fact, this is an important point which forms a part of the motivation behind this research and which we look at again in more depth as part of the literature review in Section II.

While story and narrative have many uses in SE and can be applied to various scenarios in a multitude of ways, it seems they have one primary characteristic in mind - almost universally, they are utilised as abstractions, or 'allegories'. They help the engineer understand a more complex problem, especially if the specific details of the problem are not necessarily of importance (there may be some subtleties here that are worth mentioning - as stated before, scenarios are intended to described a concrete (specific) situation to help software engineers understand the complex problem in the real-world; this, however, contrasts with stories like the Tower of Hanoi that help software engineers understand the abstracted 'computational' problem). This is not an unexpected turn of events

- stories are most often thought to simply entertain, but one of the most powerful attributes of narrative is that it helps to illuminate. The ability of humans to craft and comprehend stories is an evolutionary trait to solve societal problems [9]. It, therefore, is quite clear how Software Engineers have come to adopt various forms of storytelling; special attention has been directed to some types of story in various applications, however, the overall picture of the relationship between story and software has been largely ignored.

The motivation for this research is this exact lack of attention to the relationship between story and software. There exist various research papers on *some* of the topics that are relevant, but all focus on just one small aspect of narrative without addressing the bigger picture. Much existing research focuses simply on the use of stories as abstractions and the efficacy of this approach. While, this is something quite relevant, it forms only a small part of the whole.

The goal of this research is *not* to address Computer Science education in the manner done by Jeremy Kubica in his 'Computational Fairy Tales' [10]. We aim to dissect some possible approaches in which story can be tied to software and explore the possible conclusions or inferences that arise from this. We explore how the software engineer abstracts and transforms abstractions, how software representations/abstractions compare to those used in writing, how stories can represent software, stories as software specifications, the effectiveness of stories, etc. The hope is that, in doing so, some sort of conclusion will arise that will help illuminate the exact nature of the relationship between SE and stories. Specifically, as stated in the abstract, our 'overarching', high-level, objective is to better understand the relationship between story-thinking and computational-thinking (i.e., the relationship between how we reason with stories and how we reason with software). We split this into a series of more specific sub-objectives (SO), which are as follows.

1) Define terms such as 'story', 'story-thinking' and 'computational-thinking'.
2) Investigate several examples of story.
3) Develop a way of relating story-thinking (ST) and computational-thinking (CT) (a suite of tools).
4) Illustrate the developed tools.

As a pilot to SO3 (sub-objective 3), we look at narrative analysis as a preliminary tool and see how it may be useful in future research. At this point, it is also important to mention that, for the purposes of this research, when we say 'story' we are referring strictly to 'short stories', as opposed to something more substantive like a novel. The reason for this will hopefully become clearer towards the end of the paper, however, to summarise, a larger work akin to a novel would violate many of the definitions and constraints we propose in further sections. Furthermore, we take *'narrative'* and *'story'* to mean the same thing and may use these terms interchangeably.

In Section II, we define important terms and discuss various existing research on relevant topics and explore how they relate to our problem. In Section III we discuss the method of investigation and provide a brief overview of how and what research was performed. Sections IV- VII consider a series of worked examples which form the basis of the research. An approach to narrative analysis is considered in Section VIII and the main body of work and culmination of all research is described in Sections IX and X. Finally, we finish with a discussion on the research performed and areas for further work in Section XI.

II. Literature Review

A. What Is a Story?

Perhaps one of the most important initial questions to ask is what exactly is a story? A deceptively simple question that doesn't seem to have any one, concrete answer. In fact, it is such a difficult to answer question that it has warranted the writing of entire books such as 'The Making of a Story' by Alice LaPlante [11]. One particularly insightful section is the short essay 'What Makes a Short Story?' by Francine Prose (p. 167). Here, Prose acutely identifies the lack of a clear-cut, all-encompassing definition to describe what a story is. The essay explores the multitude of definitions others have proposed in the past, including those focusing on a certain length, perspective, plot and characters, and even that which was proposed by Aristotle himself in his Poetics [3] stating that a story must have a 'beginning, middle, and end'. Though quite logical and seemingly satisfying, each of the proposed definitions has shortcomings that Prose scrupulously explores. In the end we are left with the conclusion that a story is *most likely* something to which nothing can be added and from which nothing can be removed. Another observation Prose has made is that short stories are likely to embody Edgar Allan Poe's notion of the 'single effect', however, she also, rightly mentions that it is difficult to know exactly what this means. Although an unsatisfying conclusion, it is difficult (if not impossible) to argue with the analysis that Prose has made given her arguments and counter-arguments to the other, popular, definitions and examples. Nonetheless, if we wish to establish a formal relationship between story and software, we must have a concrete idea of what a story actually is. Therefore, for the purposes of this research we limit our viewpoint from finding a definition in the general case, to finding a good definition that works well in our context of software.

One of the most interesting definitions for the meaning of the word 'story' may, perhaps, be that which his offered by Edgar Allan Poe and his concept of the 'single effect', as mentioned by Prose. While, initially, it may be difficult to understand what exactly Poe is alluding to, thinking about his explanation of it alongside that offered by V.S. Pritchett and Chekhov may shed some light as to what it might mean. Poe says, 'in the whole composition, there should be no word written, of which the tendency direct or indirect, is not to the one pre-established design', while Pritchett and Chekhov note, respectively, 'the novel tends to tell us everything whereas the short story tells us only one thing', and 'the result is something like the vault of heaven: one big moon and a number of very small stars around it...'. To put simply, it could be said that what this 'single effect' is alluding to is the 'watering down' of a story. Each story, should ideally, have a single, clear-cut focus without many distractions. Of course, in practice, this is difficult to achieve and few readers could explain exactly what a 'single effect' is, or what precisely is the 'one thing' that our favorite story is telling us [11]. This definition, like all the others, falls short in the general case. Despite this, we propose that this is the definition which we should settle for when restraining ourselves to the context of story within software. We believe that this concept precisely defines what makes a good story with regards to software. This is due to the fact that the argument can be made that good software also embodies this principle - that is to say, it is not too 'over-engineered'; it solves a specific problem and nothing more.

We now have a suitable definition for the term 'story', however, we go one step further and consider exactly *what makes up a story*. The idea of stories being made up of 'essential elements' is not a new one and one that Prose herself mentions. There is no general consensus on what these elements are (see for example [12]), however, we propose that in addition to the definition put forward earlier, a story (in the context of software) must be composed of at least the following elements.

- Characters - the 'agents' participating in the story.
- Setting - where the story takes place (the context).
- Plot - briefly, this is the 'structure' of a story.
- Conflict - the 'problem'.
- Resolution - the 'solution' to the 'problem'.

It should be noted that in certain circumstances certain story elements may need to be inferred. For example, it is often the case that the resolution is not explicitly given in the story but we, nonetheless, have a good idea of what it might be. We revisit some of these topics in further sections and explore why our proposed definitions and constraints, not only make sense, but are a crucial step in being able to relate stories to software.

B. The Story Model

An idea that becomes very important in Section IX is the concept of the 'story model' as presented in [13]. Here, Sileno et al. propose that 'we may consider the story as a chain of events (a strictly ordered set)'. This is further extended by proposing that 'specific circumstances may be described in correspondence to the occurrence of an event'. In other words, events are associated with transitions and a set of conditions. Later, three levels of constraints are identified on the ordering of events and the strict ordering on the set of events is weakened. Sileno et al. refer to the whole composition of constraints by the term 'story-flow'.

Sileno et al. pay particular attention to the importance of plot in the story. It is the plot which imposes the series of constraints on the ordering of events in the story model, thus it is clear how the idea of plot is crucial to the structure of the proposed model. However, Sileno et al. make the remark 'Unfortunately, there are conflicting accounts about its [plot] definition in the literature. For some authors the plot coincides with how things are presented... According to an older tradition... the *story* properly said is only the chronological sequence of events, while the plot is the causal and logical structure connecting them'. It may, therefore, be prudent for the purpose of this research to define the term 'plot' in addition to the term 'story', to dissolve any ambiguity. For simplicity, we will agree with what Karin Kukkonen says in the 'Handbook of Narratology' [14]; 'The term 'plot' designates the ways in which the events and characters' actions in a story are arranged and how this arrangement in turn facilitates identification of their motivations and consequences... Plot therefore lies between the events of a narrative on the level of story and their presentation on the level of discourse'. Thus, to summarise, the plot could simply be said to deal with the *events* and *actions* of a story. This is a fitting definition as it does not stray too far from the story model proposed by Sileno et al. in their paper and, in fact, seems to be at least *similar* to the meaning of 'plot' that Sileno et al. had in mind.

The story model makes up only a small portion of the research cited, however it is of particular interest in our situation as it provides an alternative approach to thinking about stories. The proposed model is incredibly useful, however, we found that, in practice, it is fairly difficult to work with. Given a generic story, it proved quite a challenge to translate it into this model. One of the issues that arose, for example, is that there tends to be quite a bit of overlap between the *conditions* and *events* of a story. We propose that it is more beneficial to think of it in terms of *events* and *actions*, as our chosen definition of 'plot' suggests. This is especially useful when we move to the software engineering context. We revisit this again in Section IX and see how it can be useful in thinking about software.

C. Story-thinking and Computational-thinking

Perhaps, it is easiest to begin by defining the term 'computational-thinking' as this is something that most software engineers are already very familiar with. Aho [15] defines computational thinking as '...the thought processes involved in formulating problems so their solutions can be represented as computational steps and algorithms'. Put simply, computational thinking refers to the way in which we reason with and about software. It is a way of thinking that is, naturally, very efficient for SE and one that we, as software engineers, are innately comfortable with - it is, fundamentally, how we solve problems and write programs.

An approach taken by Rainer and Menon [16] is to contrast story-thinking with computational-thinking. They acutely identify the weaknesses in the traditional approach to writing software (computational-thinking); namely that it results in a 'gap' between the expectations of a stakeholder and the behaviour of the written, executable program. They introduce the idea of story-thinking as an alternative to computational-thinking. In a sense, story-thinking can be thought of as the other side of the coin to computational-thinking; put simply, it is how we reason with stories (and not software).

Rainer and Menon identify that Requirements engineering (RE) provides a bridge between stakeholders' fluid and intangible experiences, and the preparation of executable programs. They do not stop there, however, and keenly recognize that this bridge also has some limitations such as premature framing of problems. Story-thinking is a unique and clever approach to addressing some of these issues raised.

In their paper, Rainer and Menon cite work discussed by Cromer and Taggart [17] that treats story as a 'script that instructs the mind-brain to construct a mental model' - a 'situation model'. They then contrast this model with the more typical software engineering models such as a data flow diagram (DFD). They identify that there exist significant differences between the two models, however, it is suggested, quite logically, that despite these differences, story can provide a mechanism for introducing a distinct mental model into software engineering, which complements the existing models which are already accepted. In conclusion, Rainer and Menon provide a list of benefits which this way of thinking provides, which is difficult to argue with with given their arguments. This list is as follows:

1) It encourages the retention of the meaning of a situation for people.
2) It is concerned primarily with human and social aspects.
3) It encourages sense-making.
4) It retains context.
5) It discourages over-simplistic framing.
6) It encourages thinking in terms of change.

D. *Other Aspects of Story*

We briefly retrogress and consider some of the other aspects of story as well as draw any parallels that already exist between story and software. This is a matter on which there is not much existing research. One example to consider, however, is literate programming [18]. This is a programming paradigm created by Donald Knuth, who was experimenting with the concept of treating code as literature. The primary motivation behind literate programming is the improvement of the quality of code documentation. Beyond documentation, however, Knuth also makes some interesting points which may shed some light on the problem we are trying to study. In his paper, Knuth considers programs to be *works of literature* and the programmer an *essayist*. Knuth proposes that code should be embedded within documentation, not the other way around and that completed pieces of software should be publishable pieces of literature. In fact, there exist many complete examples of software which were created in this precise way. One such example is Axiom [19], a free, general-purpose computer algebra system developed by IBM. The theory behind this concept is sound and the paradigm seems to have at least *some* level of adoption. However, beyond focusing on the quality of documentation, Knuth fails to address the issue that most pieces of software do not need such extensive documentation. It would have been useful to explore the possible value of the paradigm in other areas. Furthermore, although Knuth goes into detail talking about issues such as the economic cost of adoption, he doesn't mention the complexity of adoption of such a system to the average engineer and the potential 'cumbersomeness' that it introduces.

As far fetched as it may seem, it may also be somewhat relevant to think about the parallels between software and art. Many people would consider storytelling an art form and the idea of treating computer programming as an art is not a novel one. Therefore, it is possible that if there is some link between stories, software and art, it would follow that there is a relationship between stories and software directly. This is an idea that, perhaps unsurprisingly, has also been explored by Donald Knuth in his pivotal paper 'Computer Programming as an Art' [20]. Here, Knuth goes into meticulous detail starting from the definition of the word 'art' to the comparison between art and science. As could be expected, Knuth makes the distinction that 'art' is an abstract term which can refer to many ideas. In conclusion, Knuth states that computer programming *is* an art, however, a more accurate statement would perhaps be that which Knuth himself expounds in detail in the paper - computer science is both an art *and* a science.

Understandably, there are strong opponents to the idea of code and story sharing any similarities or parallels. One interesting remark is made by Peter Seibel [21] who states that 'we don't read code, we *decode* it'. Surely this is a statement that most would agree with, however, it also begs the question whether or not this is a fair comparison. As we know, there is more to literature than just *reading* - minor differences in how something is read in comparison to a story shouldn't immediately discredit it as a piece of literature, nor should we discount any similarities it may share with stories.

Moving on to the more practical side of the link between story and software, as was mentioned briefly earlier, one of the biggest use-cases of stories here is in the form of abstractions, metaphors, analogies, etc. They illuminate a problem and help with understanding complex concepts. This is the area on which much existing research focuses. Dijkstra's 'On the Cruelty of Really Teaching Computer Science' [22] is one of the most well-cited papers in computer science in which he states that we shouldn't rely on metaphors. The paper, however, was published in 1988. It wasn't until the 1990s that learning science had become established. Today, we know constructivism as the most widely-accepted theory of how humans learn [23]. It is not out of the question that this is something Dijkstra would not have been aware of. This idea is well explored in papers such as [4], [24], [25]. These studies effectively explore the use of story in the form of metaphor and analogy in the field of SE, identifying not only the advantages, but disadvantages. Most crucially, the observations are made that there are serious drawbacks in the form of misleading metaphors, naive psychology, and even cultural differences. Interestingly, Martin Erwig [26] takes the opposite approach whereby he focuses on the use of computational concepts in story, as opposed to the other way around. These resources are immensely useful in helping us draw parallels between story and SE and in analysing how stories can be useful as abstractions for explaining more difficult computing concepts. However, their biggest drawback is that they ignore the value that stories can provide in other areas of computing - they fail to shed light on the 'bigger picture'. As mentioned previously, we are not trying to study computer science education. For example, a useful question that might have been asked is, 'is it helpful to think of code as stories' or 'to what extent can a story be used as a software specification'.

III. Method of Investigation

As part of this research, we propose an investigation comprised of a series of 'small-scale investigations' and worked examples where we attempt to shed some light on the two questions proposed in Section II ('is it helpful to think of code as stories?' and 'to what extent can a story be used as a software specification?') and, more importantly, where we try to address the objectives as outlined in Section I.

As a starting point for our investigation, we will select a series of stories and develop worked examples for each (where possible), in the hope that doing so will reveal a part of the relationship between story and software and shed some light on our questions. A summary of the selected stories is presented in Table I.

A. Extracting Stories From Source Text

The first step to working through each story was extracting the story itself. Often times, the text of interest was part of a larger work and had to be manually extracted. The source material as a whole often proved to be unusable; consider for example the Byzantine Generals [27] story, where the source material is a large and complex technical paper - the actual story itself comprises a very small portion of the overall source. This step often proved more challenging than it may initially seem. In certain situations it was not entirely clear which piece of text to extract (Hansel and Gretel). On the other hand, certain stories were already nicely separated within the source material itself and this step posed no challenge (Dining Philosophers [28]). The decision on whether to extract a verbatim version of the text or non-verbatim also had to be made for each example. In an attempt to stay faithful to the source material, the verbatim text was extracted in each case, even though this may not always have been the 'objectively' best choice.

B. Transformations

One of the most immediate observations made after extracting the story from the source material was that, in most cases, the story itself was quite unusable in a software context (that is to say, it was not immediately obvious how to go from the story to a 'reasonable' software implementation) - to get to that stage it had to undergo several transformations before ultimately becoming code.

1) The first step in the chain of transformations was to transform the extracted story into a set of coherent user stories. Although seemingly simple, this often turned out to be quite an involved task. Many of the examples looked at were composed of vague language and subtle terminology inconsistencies and contradictions which, while completely acceptable for a story, introduced a great deal of difficulty in arriving at a purely 'objective' set of user stores, and by extension, software implementation.
2) Following on, the next logical step would be to lay out the various assumptions and inferences that must be made in order to make the story more usable as a specification. Just by attempting to define the user stories, inferences had to be made about what we *think* the story is telling use in various situations. This, however, goes deeper with the need to clearly define phrases and words that are not explained in the original story. Here, already, we highlight some of the major differences identified between story and software.
3) The final (and possibly simplest) task is bringing everything together in the form of design decisions, and design documents such as class diagrams and ER diagrams. If the previous stages have been comprehensively completed, this last step is quite a trivial one, however, even here, problems arise. Considerations must be made on how to actually represent the story in the form of design diagrams and decisions. It is not always obvious how story elements map to software constructs. Furthermore, questions such as whether it is worth having classes for the sake of thematic accuracy even if they are not functional must be answered at this stage.

C. Approach to Implementing Examples

Once a story has been extracted and undergone the appropriate transformations, it should be possible to treat the story as a software specification and write up *a* specific implementation. To the keen reader, it may already be obvious that the implementation that is arrived at may be one of many possible implementations. This is possible due to the fact that stories *can* be interpreted and understood in many different ways. This fact is made even more noticeable once the story has undergone the transformations listed in the previous section as inferences and assumptions must be made at each step - these, again, will differ from person to person.

Thus, when implementing the examples in this study, it was often the case that multiple implementations were arrived at, each with a different set of assumptions and inferences. Furthermore, different approaches were experimented with for realising a sense of 'agency' in the software (i.e., characters with actions), including treating each 'character' as a class instance and an alternative approach where each 'character' or 'agent' is a separate thread. Each approach leads to slightly different results and helps shed some light on different angles of the problem being investigated.

It is also worth mentioning that, in an attempt to explore the concept of literate programming, Jupyter Notebooks were used for each implementation; this provided options to annotate, and possibly, narrate the explorations we were doing. The hope was that the value of Jupyter Notebooks, or literate programming in general, would become clearer if this was done. The full detail on technical choices is available in the 'companion' software report [31].

D. Narrative Analysis

During this research, one of the additional questions we found ourselves asking was 'how can we programmatically analyse a story?' The intention behind this was to, potentially, provide yet another angle of meeting our previously outlined objectives. To this end, work was done on a simple narrative

TABLE I
SUMMARY OF STORIES INVESTIGATED

Story	Summary
Byzantine Generals	The famous story of Byzantine Generals trying to reach agreement communicating among traitors. [27].
Dining Philosophers	The classic story depicting the problem of deadlock in threaded applications, as described by Djikstra in [28].
Hansel and Gretel	A famous German fairy tale about two siblings who were abandoned in a forest (see, for example [29]).
Pierson v. Post	A well known case in Property Law about hunting foxes and the claiming of possessions, as described in [13].
Winograd and Flores	A complex story about a broken car, taken from Winograd and Flores [30].
Shoes For Sale	A famous, six-word, short story often attributed to Ernest Hemingway, about baby shoes for sale.

analysis framework in Python. This is an unsupervised framework which employs two different approaches to performing the analysis - named entity recognition, and a more comprehensive approach which performs fully-fledged analysis using pre-trained models.

E. A Framework for Formalising Story-thinking

Until this point, the relationship between story and software has only been considered in abstract terms and in practical examples. It was realised, however that it may be helpful to try and formalise this relationship based on the examples looked at and the research employed thus far.

The third, and final, approach to investigation employed as part of this research, is the construction of a formalisation framework for story-thinking. This resulted in a substantial piece of code being written and significant research being undertaken. The creation of this framework required revisiting what exactly makes a story and attempting to, formally, define the relationship between a story and a piece of software. A Python framework was written to support this, which enables a different way of thinking about software that is more closely related to how we think about stories, while also providing functionality for visualising the 'story which is told by the software'. The development of this framework occupies a bulk of the research, however, it builds heavily on the previous approaches described earlier, and the benefits it provides may, indeed, be useful outside the context of this research.

IV. BYZANTINE GENERALS

The Byzantine Generals Problem (BGP) [27] was chosen as the first case to study in our research and served as the 'primary' example of SO2 (sub-objective 2). We say that the BGP is the primary example as it was the first story looked at; thus, from a simple time perspective, a larger proportion of efforts were devoted to studying this example than the others. Furthermore, the BGP is, arguably, the most complex of our examples, especially when it comes to the software implementations. The natural result of this is that we have more 'research' and software implementations for the BGP than for the other two stories (which are discussed in further sections).

A. Working With the Story

Extracting the story from the original paper proved to be quite a challenge. It was discovered that the BGP abstraction is actually, not only a poor story, but also an inadequate representation of the problem itself. The story we are given in the paper is intertwined with information on the engineering of the solution, which introduced a number of challenges, making it more difficult to deal with. Related to this issue is the fact that the statement of the story itself undergoes various changes and is 'updated' throughout the paper as part of the proofs and algorithms.

The difficulty of working with the BGP lies not only in the way the story is structured and presented to the reader, but also in the story itself. Consider for example the Generals that we are introduced to - we know nothing about their motives or plans; they make for poor 'characters'. Much of the same could be said about the messengers, traitors, and even the setting of the enemy city that the story revolves around. This immediately poses further questions that could be asked as part of our study - namely, 'what makes an *effective* story?', 'what is the story expressing?', and 'is the story telling us enough?'. Nonetheless, the story that was extracted is as follows.

> *We imagine that 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... All loyal generals decide upon the same plan of action... but the traitors may do anything they wish... The loyal generals should not only reach agreement, but should agree upon a reasonable plan.*

As mentioned in Section III, the above extract is a verbatim one. Initially, a non-verbatim story was extracted which we believe, would have been superior as a software specification as it was a paraphrased version of the story which was easier to parse and contained some of the assumptions and details from the engineering in the paper, such as what a reasonable plan means and what observations the generals can make. This may not seem significant, however, as will be clear later, small details such as these can have a big impact on the resulting software design and any assumptions that have to be made in a software implementation. However, in an attempt to stay faithful to the original story, a verbatim version was used instead.

Just as there was difficulty in extracting the initial story, transforming the text which was extracted into something that is usable in a software context was no trivial task (for which, in part, we can blame the inadequacy of the original story). Consider for example the (quite reasonable) user story *'As a loyal general I want to decide upon a reasonable plan of action*

TABLE II
SUMMARY OF 'BYZANTINE GENERALS' IMPLEMENTATIONS

#	Description	Classes
1	Cities are objectively 'attackable' or not; loyal generals return their true observation; traitors simply return the opposite.	General, EnemyCity, Messenger, Observation
2	As in (1) but traitors now return random observations.	—"—
3	As in (2) but cities are no longer modelled to be objectively 'attackable', i.e., each general can make different observations on the same city.	—"—
4	As in (3) but traitors now behave as in (1).	—"—
5	As in (2) but with a recursive communication algorithm from the original paper.	—"—
6	Threaded implementation of (5). Each general and each messenger exist on their own thread.	—"—

so that my division does what is best'. This seems correct, but in reality, presents us with a few problems - we don't actually *know* what the motivation of the generals is. It is not stated *why* they want to decide on a plan of action. Likewise, we don't know *how* the traitors would try to prevent the loyal generals from reaching agreement and we don't know what overarching role the messenger plays (if any at all). Further questions were raised such as what is a *reasonable* plan? What observations can the generals make? How does a traitor betray the others? These questions are just a small subset of what could be asked. Furthermore, confusion in the story's terminology such as 'observation', 'decision', and 'plan' makes working with the story a challenging task.

In a similar vein, implementing the story in code was no trivial matter. As mentioned in Section III, several implementation details were not immediately obvious at this stage and various assumptions and decisions had to be made. One of the biggest questions that had to be answered in this example is whether we should implement an *EnemyCity* class. If so, what would it do?

B. Implementation

For this example, six different implementations in Python code were arrived at. Each of these stemmed from the exact same original specification, but with a slightly different interpretation each time. This should likely be no surprise as stated in Section III. We also reiterate the fact that since this was the first (and primary) example looked at, it has more implementation 'versions' than the other examples. A brief summary of the implementations is given in Table II.

Though the differences between each implementation are quite minor, it was noticed that each produced vastly different results. For instance, in the first implementation the loyal generals *always* agreed on a 'reasonable' plan, while in others, they failed to do so consistently. However, the exact details and nuances of each implementation have been omitted for brevity.

V. DINING PHILOSOPHERS

The second example looked at is the Dining Philosophers Problem (DP) as described by Edsger W. Dijkstra in EWD310 [28] (originally proposed by Tony Hoare [32]).

A. Working With the Story

One of the most obvious differences here when compared to the BGP was how well the story was separated from the engineering. In the DP paper, there was no confusion as to what constituted the story, as was the case with BGP. Moreover, the story, once defined, underwent no further changes or modifications/updates. Thus, extracting the story was a trivial task. The extracted, verbatim, story is as follows.

The life of a philosopher consists of an alternation of thinking and eating. Five philosophers, numbered from 0 through 4 are living in a house where the table is laid for them, each philosopher having his own place at the table. Their only problem - besides those of philosophy - is that the dish served is a very difficult kind of spaghetti, that has to be eaten with two forks. There are two forks next to each plate, so that presents no difficulty: as a consequence, however, no two neighbours may be eating simultaneously.

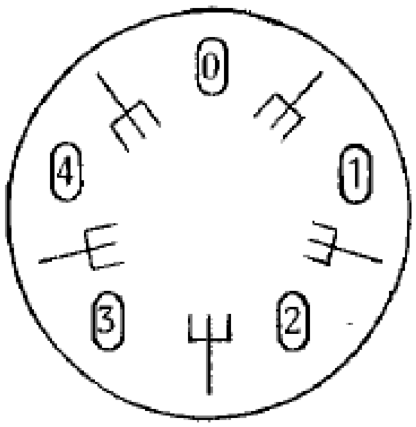

Fig. 1. Dining Philosophers table layout.

When we consider the above story, we can see immediately that it appears somewhat 'cleaner' than that of the BGP. However, when we analyse it in more depth and begin working with it in a software context we quickly realise that it suffers from some of the same issues as the BGP story. Namely, when we consider the DP as just, purely a story and disregard the software context, we see that it is not a very believable story. Once again, we know nothing about the characters or their motives. We have little information on the context of what is happening in the story. All we know is that there are philosophers and that they have trouble eating their meal. Perhaps, at this point, this is to be expected simply due to the source of the story. That is to say, the story originally was written to serve only as a means of describing a technical problem; there was no incentive to write a fully fledged story, nor was there any need to do so. In fact, as we will see in the next example (which *is* written purely as a story), having more context *does* make certain processes in the transformation chain easier, however that in and of itself introduces some difficulty.

As before, the extracted story underwent the same chain of transformations described earlier, in an attempt to make it

TABLE III
SUMMARY OF 'DINING PHILOSOPHERS' IMPLEMENTATIONS

#	Description	Classes
1	A sequential model of the story which successfully demonstrates that the problem described by the story does exist.	Philosopher, Fork
2	Threaded model of the story which demonstrates that a deadlock exists when philosophers attempt to eat with forks that are in use.	—"—
3	Solution to the threaded model based on the solution presented in the paper.	—"—

easier to work with in the software context. It is crucial to note that, despite the initial ease of extracting the story, just like in the previous example, various assumptions and inferences had to be made at each step of the transformation chain.

B. Implementation

Interestingly, with the DP, a problem that appeared early on that was not present in the case of the BGP is that it was not entirely clear what code to actually implement based on the story. The story presented was just that - a story. There was no question to be answered or explicit problem to be solved. With the BGP it was almost as if the problem itself was the story, here, however, the problem and the story are somewhat distinct. Confusion arose in that it was not entirely clear if the code should simply model the story and demonstrate that the problem does, indeed, exist, or if it should solve the problem. Nonetheless, three different implementations were written as shown by Table III.

It should be noted that the way in which the implementations were engineered required forks to be passed by reference to each `Philosopher` instance, so that they can modify the 'state attributes' of the fork (i.e. is the fork being used) and these changes would be visible to all other Philosophers. As we used Python for our implementation, the most reasonable approach for this was to simply model each fork as an instance of the `Fork` class. This, however, may not be entirely desirable from a story perspective. If we, for example, consider each class to represent a 'character' or 'agent', then it makes little sense for a fork to be a class. This highlights an interesting peculiarity that we have not seen before in that the implementation of a story, to a certain extent, depends on the nuances of the chosen language and platform.

C. A Note on Implementation Constraints

The third implementation (as described in Table III) is likely the most interesting of the three as it actually solves the problem described in the paper. Unlike with the algorithm presented in the BGP paper, the algorithm we are given here was logical to follow and easy to understand, and was trivial to implement in Python (see [31]). Likewise, the result was exactly as expected and described in the paper - something we had difficulty achieving in the case of the BGP. One issue that arose, however, is that with the introduction of state variables and semaphore objects, it could be said the the implementation stepped outside the boundaries of the original story. The code worked flawlessly, but it no longer was a 'faithful' representation of the specification as we were now modelling constructs and behaviour that was not necessarily described in the original story. This raises issues regarding what the line is between story and implementation and whether we are permitted to cross this line. In this case, we propose that the original specification/story be amended by adding the following piece of text.

> *The philosophers overcome this problem [of eating spaghetti] by signalling amongst themselves whether or not it is appropriate to eat. When a philosopher lets the others know that he is hungry, he checks to make sure that none of his neighbors are eating before picking up his forks and consuming his meal. Once he is finished, he signals to his immediate neighbors that he is no longer eating. The philosophers next to him can then follow the same procedure should they get hungry. Thus, the philosophers ensure that no two neighbors attempt to eat their meal simultaneously.*

This justifies the design decisions taken, however, it is not entirely clear whether making changes such as this is (should be) within our power as software engineers. Should we be entirely faithful to the original story at the cost of implementation? One of the strengths (and weaknesses) of stories is that they can be interpreted in different ways. Perhaps, this property suggests that we don't even need to amend the original specification - so long as the general 'theme' of the software corresponds to that of the story, we are not breaking any rules.

VI. HANSEL AND GRETEL

Moving on, we consider a different kind of example - that is to say, we look at a story which was written purely as a story, with no relationship to software engineering. Namely, we consider the famous German fairy tale 'Hansel and Gretel' (HG), collected by the Brothers Grimm (see [29] for one possible version of this).

A. Working With the Story

To the reader, some issues regarding the chosen example may already be obvious. You may be thinking to yourself how could a fairy tale such as this be implemented in code. In fact, this was an immediately apparent issue when it came to working with this example. There was challenge at the very first (and, arguably, the most important) step - extracting the story. We couldn't simply follow the same procedure as before; there was no immediate piece of text that could be treated as the story because the entire text was a story. The initial idea was to not extract anything at all and just treat the entire piece of text as our 'specification', however it was not entirely clear how this would work or whether it was even feasible. Thus, with a a little inspiration from Erwig [26], it was decided that the extracted story would be that of the description of Hansel's clever 'path-finding' algorithm to find his way out of the forest. We say that we were inspired by Erwig because this is precisely the part of the story that he focuses on in his 'Once Upon an Algorithm'. This choice does make sense, however, for multiple reasons. For one, it could be said that Hansel's 'exploit' to trick his stepmother and find his way out of the forest is the 'essence' of the HG story. Not only because it makes up the bulk of the story physically but also because it is one of the most widely recognisable 'traits' of HG that most people who know the original are familiar with. Moreover, this is a good choice because (as Erwig notices) it represents a computing concept (path finding) that can be 'easily' implemented in code. Without further digression, the extracted story is given below.

Hansel and Gretel were scared. They knew the forest was deep and dark, and that it was easy to get lost. 'Don't worry! I have a plan!' whispered Hansel to Gretel. He went to the back of the house and filled his pockets with white pebbles from the garden. Then the two children started walking, following their stepmother's directions. Every few steps, Hansel dropped a little white pebble on the ground. ... Hansel waited until the moon was bright. The moonlight shone through the tall trees and made his tiny white pebbles glow. They followed the trail of pebbles all the way back home.

It is interesting to note, that when the story underwent the chain of transformations from user stories, to assumptions and design decisions, fewer inferences had to be made than usual. Of course, assumptions still had to be made regarding the design decisions and specific implementation nuances. However, consider, for example, the user stories - if we have a user story 'as Hansel I want to follow my dropped pebbles to get back home'; this is a reasonable user story but, more importantly, notice how it contains no assumptions or inferences. All the information required to form the user stories is found within the original text - something that is not true for the previous two examples. Thus, it could be said that due to the additional context we have for this example, transforming the extracted story was more trivial and less prone to inferences and assumptions.

B. Implementation

Once the story had been extracted, it was not immediately obvious how to implement it. It was, however, quite clear from the story what to do - it was clear that a path-finding algorithm of sorts was required. In this regard, the HG story was (surprisingly) more clear than that of the Dining Philosophers. However, at this step many 'large' design decisions had to be made in order to arrive at a suitable implementation. This is something that was not experienced in the previous examples. In the previous two examples, many assumptions and inferences had to be made at the design stage, however the implementation details were often fairly clear; here, however, we seem to have the opposite situation. There is certainly something to be said of the fact that the relationship the previous two examples had to software engineering problems made the implementation easier. With HG, on the other hand, the benefits of utilising a complex, fully-fledged story as a software specification (lack of assumptions/inferences at the design stage) are counter-balanced by a trickier implementation stage. It seems that when designing a story, there is the challenge of providing just the right context (and amount of context) so that the design *and* implementation stage can be completed with a minimal amount of assumptions and inferences. This is tricky, because we don't want to have a story that is *too complex* (as we will see later). Moreover, we also don't want to sacrifice the very qualities and attributes that make a story - namely, that it *can* be interpreted in different ways as this is both an advantage and a disadvantage.

TABLE IV
SUMMARY OF 'HANSEL AND GRETEL' IMPLEMENTATIONS

#	Description	Classes
1	Hansel's 'path-finding algorithm' is implemented in 2D space.	Hansel, Pebble
2	Abstract implementation of (1) which works for arbitrary graphs described by a set of nodes and edges.	-

The approach that was eventually settled for was to model Hansel as existing in 2D space where each of his pebbles dropped had an *x* and *y* coordinate. With this design in place, Hansel would simply walk forwards towards his destination, and to turn back home, he would follow the pebbles he dropped. This is based on the location of the pebbles and on which pebble is visible from a given point - there is no 'cheating' going on by simply iterating over the collection of pebbles in reverse order.

A further implementation was then written up where an attempt was made to make the initial implementation more 'abstract'. This proved to be quite a success - the result was a single, short method which accepts any generic graph described by a set of nodes and edges. Each node represents a 'pebble' and an edge represents the visibility of pebbles. Thus, it is possible to traverse the graph to get from a starting point to an end point. Interestingly, the result of this implementation is not consistent with that of the original. This is likely due to the ordering of the 'pebbles' or 'nodes' and the sequence in which they are traversed. It is possible to extend this implementation even further by adding the option for Hansel to backtrack (so as to avoid getting stuck) and even prioritizing edges based on distance, however then we run into the same issue of the boundary between story and implementation as discussed in Section V. The two implementations are briefly summarised in Table IV.

VII. OTHER EXAMPLES

Here we go over a few 'difficult' stories and try to identify why they fail as examples. These examples are not strictly 'more challenging' than the previous examples, however they deserve a section of their own as these are three examples for which no coded implementation was attempted - each of these was taken only as far as the design stage.

A. Winograd and Flores

The first example we look at is taken from Winograd and Flores [30].

You have been commuting to work in your old Chevrolet. Recently you have had to jump-start it three times, and there has been an ominous scarping sound every time you apply the brakes. One morning as you are driving to work you cannot get it into first gear. You take it to a mechanic who says there is a major problem with the transmission. ...You talk to your husband and decide that there are several alternatives – you can have the old car repaired, or you can buy a used or a new car. If you want a used car you can try to get it through friends and newspaper ads, or you can go to a dealer. If you get a new one you may want a van you can use for camping trips, but you're not sure you can afford it and still go on the vacation you had planned. In fact you're not sure you can afford a new car at all, since you have to keep up the payments and insurance on your husband's car as well. ...on the next day (since you can't drive to work) you call and check the city buses and find one you can take. After a few days of taking the bus you realize you didn't really need the car... ...[after a few days of commuting on the bus, you realise that the] bus ride takes too long, and you are complaining about the situation to your friend at work. He commiserates with you, since his bicycling to work is unpleasant when it rains. The two of you come up with the idea of having the company buy a van for an employee car pool. ...[later, you] talk to another friend who has just gotten his car back from the shop. He hears your story and expresses surprise at the whole thing. It never occurred to him to do anything except have it fixed, just as he always did.

In this story we are presented with a variety of different 'scenes', each with a series of different characters. As stated by Rainer and Menon [16], each scene seems to *dissolve* the problem rather than solve it. They mention how each particular scene may itself be treated as a unique scenario, with its own, individual set of user stories, design diagrams, and a software solution. In fact, this is exactly what we observed when taking this story to the design stage. The result was a series of potential software implementations, each tackling a specific problem in the story such as how/where to buy a car, where to find a mechanic, how to find a bus, etc. If any of these software implementations was pursued, we would, essentially, be simplifying the original context or, as Rainer and Menon put it, de-contextualise. This is a very complex story which presents us with an interesting challenge that we have not observed in the previous example. For these reasons, we propose that there exists an 'upper-bound' on the complexity and detail of a story when working with the software context. A story that is too complex, simply cannot be efficiently translated into software.

B. Shoes For Sale

The second example we look at is a famous six-word, short story often attributed to Ernest Hemingway.

'For Sale. baby's shoes. Never worn.'

This short story is on the opposite end of the complexity spectrum, relative to the previous example. For us humans, an incredible amount of information can be inferred from just these simple, six words. This is an observation made by Rainer and Menon [16] where they state that 'one surface-level problem can be inferred, i.e., the problem of selling a pair of shoes. Yet there are deeper questions provoked by this story, e.g., what happened to the baby, how do the seller and baby relate, and why is the seller selling'. This is indeed fascinating when considering this example as purely a story, however, in the software context, the reality is that there is simply not enough information to do any meaningful work on it. A software implementation based on this story is, for all intents and purposes, impossible. In fact, the design stage itself proved to be quite a challenge of which the outcome was not entirely valuable. With the previous example, we considered how a story can be too complex to be implemented in software, yet here we have the other side of the coin - the story is not complex enough. Thus, as before, we propose that just as there is an 'upper-bound' on the complexity of a story, there is also a 'lower-bound'.

C. Pierson v. Post

The third, and final, example is the well known case in Property Law: 'Pierson v Post (1805)', as described in [13]. The summarised story is as follows:

Post was hunting a fox with a horse and hounds in a wild and uninhabited land, and was about to catch it, but Pierson, although conscious of Post's pursuit, intercepted, killed and took the animal. Both claimed the fox, the first appeal had found for Post, but this court reverted the previous result. The different positions are expressed by two judges: Tompkins (majority) and Livingston (dissent). The first, supported by classical jurisprudence, claims that possession of a fera naturae, where fera naturae is an animal wild by nature, occurs only if there is occupancy, i.e. taking physical possession. Pierson took the animal, so he owns it. The second argues that if someone starts and hunts a fox with hounds in a vast and uninhabited land has a right of taking the fox on any other person who saw he was pursuing it.

This is an interesting example because it may not be immediately obvious why it fails in the software context. It may be quite clear that a software implementation of the story is likely to be challenging, or even impossible, but it is certainly not clear why that is. The story has a 'good' length, a problem to be solved, it is not too complex, and it is not too 'simple', so what is the problem? While it is true that the story is, for all intents and purposes, a good candidate for a software implementation at first glance, in this specific example, it is likely that the story is simply 'too abstract'; it is not clear what a software solution would even do or what *explicit* problem it would solve. A computer program cannot perform the role of the judges in this scenario, nor is there any need for it to do so. There is simply no value in trying to represent this solution in software, nor is it feasible. Based on this, we propose that in addition to the, already defined, constraints on complexity, some stories simply are not 'computationally representable', or there is no value in doing so.

VIII. Narrative Analysis

Shifting our focus a little and looking at the problem of stories and software from a different angle, we find ourselves asking the question 'how can we programmatically analyse a story?'. This is an approach that we have not discussed yet which, nonetheless, may be provide some valuable insights into the relationship between story and software. The idea was that by analysing a story and considering, in depth, what makes up each individual story, we may be able to relate the findings back to the software context.

The approach to this was via narrative analysis, by way of a series of Python programs. The primary inspiration for this approach is the excellent paper by Min and Park [33] which models various components of narrative such as character interactions and performs analysis such as topic modelling and sentiment analysis for Victor Hugo's Les Misérables [34].

A. Named Entity Recognition

The first approach to narrative analysis involved named entity recognition (NER). The idea was that once the NER was performed and the entities extracted, a model could be developed representing the relationships between each of these entities. The end result of this could be something along the lines of an ER diagram. This would, of course, provide a direct visual representation of the link between a story and the software implementation. It could be thought of as a 'pre-ER' step. Perhaps it could be useful for comparing with the actual ER diagram which gets drawn up during the development stage. Ideally, this could be taken even further by analyzing the text to try and extract any 'action' that each entity performs, providing even more insight into how this would relate to a software implementation. In practice, however, the results of this were disappointing. NER is not as accurate as expected, and in most of the examples, the named entities of a story do not correspond as nicely to software objects as anticipated. For example, thinking about the BGP, it would have been useful to receive an output such as *[General, Messenger]*, but this was not the case during experimentation.

B. Further Narrative Analysis

Following from the above, a more 'in depth' approach was taken where, instead of simply relying on NER, comprehensive analytic methods were employed, utilising pre-trained models to perform supervised narrative analysis (see [31] for full technical details). The results of this approach were far more promising.

In this implementation, instead of relying on extracting named entities, it was opted to extract nouns instead. This, of course, is a much coarser approach that can yield wildly inaccurate and 'messy' results that largely vary from text to text, however, the results proved to be acceptable in most situations. Various methods for filtering 'bad' nouns were introduced which can be varied on a case-by-case basis with the help of a comprehensive command line interface. This allows for enough granularity such that each story, no matter how complex, can provide good results.

Once the nouns had been extracted, the next step was to determine the relationships between each noun in the original story. A crude method of doing this was implemented which simply determines which nouns occur together in a sentence - if noun A occurs in the same sentence as noun B, there is said to be a relationship between A and B. It is likely that there is a better and more fine-grained alternative to this approach, however, this implementation has, once again, surprised with the quality of its results.

With the nouns having been extracted and the relationships between them established, a node graph is produced which visualises the parsed data. Example output for the narrative analysis of the BGP story is shown in Figure 2. This is more useful than may initially seem as the graph, essentially, provides us with a visual representation of the 'actors' in a story and how they interact with each other. Of course, not every noun will correspond to an 'actor' or 'character', however, for most situations, the results are definitely acceptable at the very least. This narrative analysis may even be considered as an additional step in the design stage of implementing a story in software. This is useful not only for comparing with the final implementation, but also as a guide in designing the software itself.

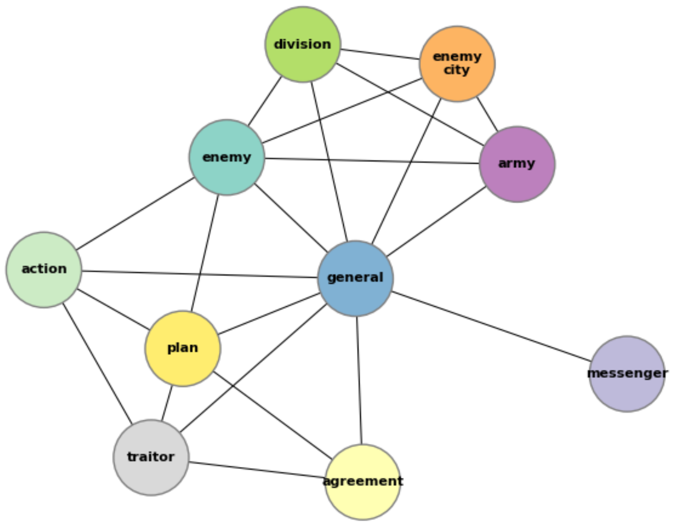

Fig. 2. Example narrative analysis of BGP.

C. Extracting Verbs

One of the earlier ideas for expanding the narrative analysis was the extraction of verbs for each noun, and, while not much work has been completed on this front, it may be of interest to at least briefly mention it here. Extracting verbs for each noun would allow better relationships to be drawn up, and even the creation of a 'pseudo' class diagram. This functionality would have been incredibly powerful as it would allow, not only the extraction of 'actors', but also their actions. This would significantly narrow the gap between a story and its software implementation.

Unfortunately, the complexity of this proposed concept spans far outside the scope of this research. Several experiments were performed to gauge the viability of this idea, however they were all extremely disappointing. Thus, this idea was not pursued further.

IX. Story-thinking in Software Engineering

In Section II we defined key concepts such as 'story-thinking' and 'computational-thinking', and explored the meaning of the term 'story' as well as considering what exactly makes up a story. In subsequent sections, we conducted a number of investigations of stories. Now that we have addressed sub-objectives 1 and 2, we want to consider some possible ways of relating story-thinking and computational-thinking as per SO3. Despite having carried out all the prerequisite work, this can be quite a challenging task; in fact, the theory we present in this section was, initially, something quite unexpected and somewhat surprising. Nonetheless, we now propose that any story which can be defined with the definitions put forward in Section II and which doesn't violate the constraints set out in Section VII, can be mapped directly to a suitable software implementation.

A. Elements of Software

We now revisit some topics mentioned earlier in Section II. We pay particular attention to the previously defined elements of a story and the proposed 'story model'. Upon closer investigation, we can see how the elements of a story may provide us with a way to relate story to software. Through a careful examination of the three main stories looked at before, their software solutions, and the elements that make up each story, we observed that it may be possible to adopt a similar approach to breaking down the software component into a set of basic and generic components. Through various iterations and experiments we see that each story element has a near-one-to-one equivalent in the software context. Thus, we propose that a piece of software, similar to a story, can be described by a series of 'basic' elements. These are as follows.

- Agents - the primary actors of the software solution. This is where most of the logic and 'executable' code will reside. The agents are generally derived directly from the characters of the story, however, this may not always be the case.
- Events - these are global events that occur in a world and can trigger other events.
- Actions - actions are similar to events, however, they are not global but local to the agents.
- Components - miscellaneous elements of the software solution that make up neither the agents nor the events/actions, but are nonetheless, integral to the software solution. Generally, the components are what tie the agents, events, and overarching software solution together.

TABLE V
Mapping of Story Elements to Software

Software Element	Derived From
Agents	Characters
Events	Plot (Events)
Actions	Plot (Actions)
Components	Setting, Conflict, Resolution
Flow	Plot

- Flow - this describes the interaction between the events and the agents/actions. This is done by what we refer to as Flow Description Language (FDL).

These elements can, generally, be derived directly from the elements of the story. They may not always be, however, direct, one-to-one equivalents. Table V shows one possible approach to deriving software elements from story elements.

B. Flow Description Language

The mapping of story elements to software elements is only one half of the whole. This element mapping simply provides a way of *describing* software, however, software must also, naturally, perform logic and solve problems. This is where the concept of 'plot' and the 'story model' as defined in Section II play a central role. With these two ideas we can relate the 'story-flow' (as put forward by Sileno et al. [13]) to the *'software-flow'*.

It should be clear now why our proposed definition of 'plot' and the slight modification to the story-model to focus on events and actions, rather than conditions and events, is beneficial - events and actions is a more natural and logical way to describe software than events and conditions. We are more interested in what is happening in the story than the conditions that must be met for those things to happen.

The term 'flow' describes the interaction between events and actions. We have developed a very simple 'markup' language for describing these interactions - Flow Description Language (FDL). This, essentially, provides a high-level, textual overview of what the software component does in terms of events and actions. Every piece of software can, fundamentally, be described with FDL when it is modelled in terms of our proposed software elements. Besides acting as a textual representation of the functionality of a software component, FDL can also be helpful as a design-stage tool to help plan out a specific software implementation, however, it's dynamic nature means that it will likely need updating as software is written and iterated upon. The basic syntax of FDL is as follows.

```
event|action -> {events|actions...}
```

To simplify, on the left of the arrow we have a single event or action, acting as the 'source' or 'origin', and on the right of the arrow we have a set of events or actions that are 'triggered' by the 'origin' on the left. This flow between events and actions is, in theory, how the software is able to carry out its logic.

C. A Framework for Story-thinking

The realisation that there may be a deeper connection between story and software led to the development of a story-thinking framework where we are ably to apply the above theory in practice. This is a Python library which builds upon the proposed definitions and constructs, to allow for a different way to approaching software development - one that is more in line with how we reason with stories (and not software). As such, this Python library lessens the gap between story-thinking and computational-thinking.

When working with the proposed Python library, we model software as being made up of a global *world* instance; this, in turn, is composed of *agents* and *components*. Each agent can perform actions and the world itself can emit global events. Events and actions can trigger further events and actions - this is how the software carries out its logic. With this approach, we are able to structure software more similarly to how a story would be structured, thus we are able to introduce some degree of story-thinking into computational-thinking. Currently, the Python library also supports threading (we can model agents/characters as instances of a class or as individual threads), automatic FDL generation, automatic finding of 'origin' events, live-stepping through events and actions, and the visualization of, both, the FDL and the software execution trace itself (in terms of actions and events) in the form of node graphs. Functionality is also exposed via a command line interface and a simple web based interface. We look at some examples of this in the following section.

Most of the functionality of the Python library can be achieved via class inheritance and/or function decorators [35]. This is quite 'non-intrusive' in that existing software can be re-written to make use of the library without too much work. This allows for easier adoption of, not only the proposed Python library, but also of this different and unfamiliar approach to writing software. As mentioned in Section II, computational-thinking is very efficient for writing software and a 'story-thinking approach' does require the software developer to think differently about their software design - we now have to think primarily in terms of *events* and *actions* - this is not always trivial.

X. Demonstration of Story-thinking

To demonstrate how story-thinking and our proposed Python library may be applied in practice, we consider the BGP as an example.

```
agents: [General]
events: [camp, make_order]
actions: [General.speak, General.listen, General.make_observation]
components: [Observation, Decision, Plan, EnemyCity]

flow:
– camp –> {General.make_observation}
– General.make_observation –> { }
– make_order –> {General.speak}
– General.speak –> {General.listen}
– General.listen –> {General.speak}
```

Listing 1. BGP software elements.

```
characters: [Generals]
setting: ['Byzantine army', 'Enemy city']

plot:
– Army divisions camped outside enemy city
– Generals share their observations
– Traitorous generals preventing loyal
  generals from reaching agreement

conflict:
  Communication between loyal generals
  among generals which are traitorous

resolution:
  Reliable means of communication
  for loyal generals
```

Listing 2. BGP story elements.

```
from formalisation import World, Agent

w = World()

class General(Agent):
    @w.event('camp')
    def make_observation
        ...

    @w.event('General.listen')
    def speak
        ...

    @w.event('General.speak')
    def listen
        ...

@w.event()
def camp
    ...

@w.event()
def make_order
    ...

if __name__ == '__main__':
    w.process()
```

Listing 3. BGP psuedo-code example of the story-thinking Python library.

Listing 2 shows us the story elements that make up the extracted story for the BGP, while Listing 1 shows the corresponding software elements of the respective BGP implementation. We visualise this with YAML documents.

Now we consider Listing 3; here we can see how the previously defined story/software elements may translate to code with the help of our proposed Python library. From looking at the pseudo-code we can see immediately how the software component is composed of the elements that were derived from the corresponding BGP story.

One of the 'side-benefits' of utilising our Python library is the ability to visualise the interactions between events and actions. This, in turn, helps us visualise the underlying algorithms that make up the software solution. This is useful for a plethora of reasons such as helping us understand complex software and for debugging purposes. An example of this is shown in Figure 3.

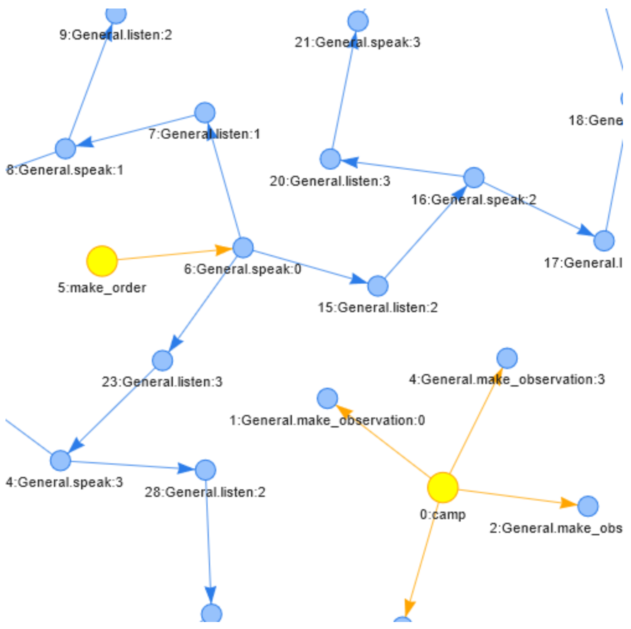

Fig. 3. Example visualisation of BGP implementation (zoomed in).

XI. Discussion

A. Findings and Conclusion

In this research we propose definitions for various terms such as 'story', 'plot', 'computational-thinking', and 'story-thinking' (SO1). We also introduce, and slightly alter, the 'story-model' as originally presented by Sileno et al. [13]. These constructs and definitions lay the foundation for relating stories with software. We investigate a series of stories and demonstrate the challenges involved in moving from a story to a software implementation; we highlight the processes and transformations that a story must undergo before it can become code (SO2). Based on the investigation into the various stories, and their coded implementations, we see that *both* story and software can be broken down into a series of elements. These elements, together with the story-model, provide a way to map story to software. We also identify a series of constraints and limitations to the stories that can be worked with in the software context.

Based on the findings we see that there is indeed a deeper link between stories and software and that stories are not necessarily limited to being used as metaphors or abstractions; an idea that is well explored by research such as [4], [24], [25]. We observe that, in fact, code and stories may not be so different after all as is proposed by Seibel [21]. By building on and expanding on the earlier mentioned story-model and the idea of story-thinking as proposed by Rainer and Menon [16], we were able to meet the final two sub-objectives laid out in Section I (SO3, SO4) - that is, we develop a way of relating story-thinking and computational-thinking. Thus, we lessen the gap between how we reason with stories and how we reason with software. We also shed light on the two questions proposed at the end of Section II; we see that it, indeed, *may* be helpful to think of code as stories and we explore the limitations to the extent with which a story can be used as a software specification.

We observe one of the biggest strengths and weaknesses of story - the fact that they can be understood in many different ways. The biggest implication of this is that inferences and assumptions made must be clearly defined; this can lead to several different software implementations. Though initially we may be led to believe that there is a small, or insignificant, link between story and software, by developing a story-thinking framework in Python, we see, as already established, that this is not the case. Through the proposed Python library and a 'story-centric' approach to developing software, we are able to implement at least *some* aspects of story-thinking in computational-thinking. We believe this marks the first step towards implementing story-thinking in software and thus realises some of the benefits discussed in Section II such as the retention of context and better meeting stakeholder expectations. There are also some 'side-benefits' to the visualisation aspect of the library. It is quite clear to see how the visualisation can help in the understanding of complex software and even debugging. Moreover, an interesting discovery is that the visualisation provides us with a 'physical' representation of what the code is doing. We could think of this as the 'story the code tells'; this provides a unique and interesting link between the software component and the initial story specification.

B. Limitations

The proposed library and framework for thinking about and approaching software development can be used for generic programs - it is not constrained to the examples we provide. It is straightforward to use, however, one of the biggest challenges is that it involves a different way of thinking about software. For example, one of the problems encountered when working on examples is what to do when action *B* needs to be called from action *A*, but *before* action *A* completes. This, of course, introduces much development overhead and may result in unexpected software behaviour. It is likely that many such edge cases and complications exist and there is certainly much work remaining in terms of refining the proposed framework.

It is also possible that the theory proposed in Section IX may need some minor revisions. Specifically, we think about the software elements - there *may* be potential to remove some non-crucial elements and even add some others. Consider, for example, the 'components' element; this is an element that we believe is not as strongly defined as the others and its importance is not entirely clear.

Furthermore, our method of relating story-thinking to computational-thinking relies heavily on our proposed definitions and concepts such as the story-model. We also introduce a series of constraints on the story when moving from story to software. Thus, if a story violates these constraints or steps outside the boundaries of our proposed definitions, we make the assumption that a software implementation is not possible (see Section VII). However, this may not always be the case. Consider for example the case on the complexity of a story - we set an upper bound and a lower bound to this, however, it may be the case that if a story is *too* complex (as was the case with the Winograd and Flores [30]), we can simply break it down and treat it as several 'sub-stories'. This is something that has not been considered in depth.

C. Where Next?

One of the most important next steps is to apply the story-thinking framework to a wider range of examples and study each one carefully to try and quantify the real value that story-thinking provides to software engineers. There may be potential in areas such as retrieving lost software specifications (by generating FDL and software visualisations) and an approach to objectively compare different pieces of software (for example, by applying graph algorithms to the output generated by the library). While we have discussed the benefits of story-thinking, benefits do not necessarily equate 'utility' - especially when it comes to application in industry. We have also briefly looked at an approach to narrative analysis, and identified this as a potential avenue for future research as this may provide a different angle of relating stories to software. Above all, this research was focused on deepening our understanding of the relationship between story and software, and we feel we have succeeded in this regard, however, this leaves one burning question that needs answering and which marks the most important next step for future research - is this useful?

References

[1] E. Andrews, "What is the oldest known piece of literature?" [Online]. Available: https://www.history.com/news/what-is-the-oldest-known-piece-of-literature

[2] S. N. Kramer, "Sumerian Literature; A Preliminary Survey of the Oldest Literature in the World," *Proceedings of the American Philosophical Society*, vol. 85, no. 3, pp. 293–323, 1942. [Online]. Available: http://www.jstor.org/stable/985008

[3] Aristotle, *The Poetics of Aristotle*, Nov. 1999, [Butcher, S. H. (Samuel Henry)]. [Online]. Available: https://www.gutenberg.org/ebooks/1974

[4] M. Forišek and M. Steinová, "Metaphors and analogies for teaching algorithms," in *Proceedings of the 43rd ACM technical symposium on Computer Science Education - SIGCSE '12*. Raleigh, North Carolina, USA: ACM Press, 2012, p. 15. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2157136.2157147

[5] B. Gavish and G. Stephen C., "THE TRAVELLING SALESMAN PROBLEM AND RELATED PROBLEMS," *Massachusetts Institute of Technology*, p. 33, Jul. 1978.

[6] P. Cull and E. F. Ecklund, "Towers of Hanoi and Analysis of Algorithms," *The American Mathematical Monthly*, vol. 92, no. 6, pp. 407–420, Jun. 1985, publisher: Taylor & Francis _eprint: https://doi.org/10.1080/00029890.1985.11971635. [Online]. Available: https://doi.org/10.1080/00029890.1985.11971635

[7] D. Leffingwell and P. Behrens, "A User Story Primer," *Leffingwell, LLC.*, p. 16, 2009. [Online]. Available: https://appliedframeworks.com/wp-content/uploads/2019/12/user-story-primer.pdf

[8] J. Carroll, M. Rosson, G. Chin, and J. Koenemann, "Requirements development in scenario-based design," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, pp. 1156–1170, Dec. 1998, conference Name: IEEE Transactions on Software Engineering.

[9] A. Parry, "Why We Tell Stories: The Narrative Construction of Reality," *Transactional Analysis Journal*, vol. 27, no. 2, pp. 118–127, Apr. 1997. [Online]. Available: https://www.tandfonline.com/doi/full/10.1177/036215379702700207

[10] J. Kubica, "Computational Fairy Tales: Books." [Online]. Available: https://computationaltales.blogspot.com/p/book.html

[11] A. Laplante, *The Making of a Story: A Norton Guide to Creative Writing*, reprint ed. New York: W. W. Norton & Company, Jan. 2010.

[12] C. R. Duke, "Teaching the Short Story," *The English Journal*, vol. 63, no. 6, pp. 62–67, 1974, publisher: National Council of Teachers of English. [Online]. Available: https://www.jstor.org/stable/813432

[13] G. Sileno, A. Boer, and T. Van Engers, "Legal Knowledge Conveyed by Narratives: Towards a Representational Model," p. 10 pages, 2014, artwork Size: 10 pages Medium: application/pdf Publisher: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2014/4656/

[14] K. Kukkonen, "Plot," in *Plot*. De Gruyter, Oct. 2014, pp. 706–719. [Online]. Available: https://www.degruyter.com/document/doi/10.1515/9783110316469.706/html

[15] A. V. Aho, "Computation and Computational Thinking," *The Computer Journal*, vol. 55, no. 7, pp. 832–835, Jul. 2012. [Online]. Available: https://academic.oup.com/comjnl/article-lookup/doi/10.1093/comjnl/bxs074

[16] A. Rainer and C. Menon, "Story-thinking and computational thinking in human-centric software engineering," in *(Submitted to) CHASE '22: Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE)*, Pittsburgh, PA, USA. ACM, New York, NY, USA, 2022, p. 5.

[17] C. Comer and A. Taggart, *Brain, Mind, and the Narrative Imagination*. Bloomsbury Publishing, Jan. 2021, google-Books-ID: otkOEAAAQBAJ.

[18] D. E. Knuth, "Literate Programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, Jan. 1984. [Online]. Available: https://doi.org/10.1093/comjnl/27.2.97

[19] T. Daly, "daly/axiom," Nov. 2021, original-date: 2008-02-27T23:58:50Z. [Online]. Available: https://github.com/daly/axiom

[20] D. Knuth, "Knuth: Computer Programming as an Art." [Online]. Available: http://www.paulgraham.com/knuth.html

[21] P. Seibel, "Code is not literature," Jan. 2014. [Online]. Available: https://gigamonkeys.com/code-reading/

[22] E. W. Dijkstra, "On the cruelty of really teaching computing science (EWD 1036)," *The University of Texas at Austin*, Dec. 1988. [Online]. Available: https://www.cs.utexas.edu/~EWD/transcriptions/EWD10xx/EWD1036.html

[23] M. Guzdial, "Dijkstra Was Wrong About 'Radical Novelty': Metaphors in CS Education." [Online]. Available: https://cacm.acm.org/blogs/blog-cacm/248985-dijkstra-was-wrong-about-radical-novelty-metaphors-in-cs-education/fulltext

[24] J. D. Herbsleb, B. Laboratories, and S. Boulevard, "Metaphorical Representation in Collaborative Software Engineering," p. 10.

[25] J. P. Sanford, A. Tietz, S. Farooq, S. Guyer, and R. B. Shapiro, "Metaphors we teach by," in *Proceedings of the 45th ACM technical symposium on Computer science education*. Atlanta Georgia USA: ACM, Mar. 2014, pp. 585–590. [Online]. Available: https://dl.acm.org/doi/10.1145/2538862.2538945

[26] M. Erwig, *Once Upon an Algorithm: How Stories Explain Computing*. Cambridge, Massachusetts ; London, England: MIT Press, Sep. 2017.

[27] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, p. 20.

[28] E. W. Dijkstra, "Hierarchial Ordering of Sequential Processes," p. 29, 1971. [Online]. Available: https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF

[29] J. Grimm and W. Grimm, "Hansel and Gretel," in *Children's and Household Tales – Grimms' Fairy Tales*, 7th ed., 1857, vol. 1, pp. 79–87. [Online]. Available: https://sites.pitt.edu/~dash/grimm015.html

[30] T. Winograd, F. Flores, and F. F. Flores, *Understanding Computers and Cognition: A New Foundation for Design*. Intellect Books, 1986, google-Books-ID: 2sRC8vcDYNEC.

[31] K. Mosoczy, "'Story and Software' Software Development Report (companion software report)," 2022. [Online]. Available: https://github.com/konmos/csc4006/raw/master/docs/software_report.pdf

[32] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978. [Online]. Available: https://doi.org/10.1145/359576.359585

[33] S. Min and J. Park, "Modeling narrative structure and dynamics with networks, sentiment analysis, and topic modeling," *PLOS ONE*, vol. 14, no. 12, p. e0226025, Dec. 2019, publisher: Public Library of Science. [Online]. Available: https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0226025

[34] V. Hugo, *Les Misérables*, Jun. 2008. [Online]. Available: https://www.gutenberg.org/ebooks/135

[35] K. D. Smith, J. J. Jewett, and A. Baxter, "PEP 318 – Decorators for Functions and Methods | peps.python.org," Jun. 2003. [Online]. Available: https://peps.python.org/pep-0318/