Queen’s University Belfast

School of Electronics, Electrical Engineering and Computer Science

CSC4006 Software Development Report

“Story and Software”

Konrad Mosoczy

40224461

1 Contents

2 Introduction

This software development report is the counterpart to the research report for my project "Story and Software", or more verbosely, "Story Thinking and Computational Thinking in Software Engineering". Simply put, this is a project which aims to analyse the relationship (if any) between how we reason with stories and how we reason with software. As is likely already obvious, this is a very unique problem and one that has not been looked at extensively in the past. It is a very research-heavy project, and it may even be difficult to imagine how/what software could be written to shed light on such a problem. Thus, the software written as part of this project is not the typical piece of software where there is a single system, a single specification, and a single focus. Indeed, in the end there was a single piece of software which ended up being the primary focus of the project, however, before reaching that point, various individual pieces of software were written – each of these entirely separate yet integral to solving the overarching problem. As such, this report will be structured to reflect this "peculiarity" and we will describe each of the individual software "components" in turn.

Initially, the relationship between "story thinking" and "computational thinking" was analysed by way of a series of worked examples. For each of these examples, we start off with a traditional "story" and transform it such that it can be usable as a software specification. This involves defining assumptions, inferences, creating design diagrams, etc. This was then used to "implement" the story in code. Each worked example had to undergo various iterations, ultimately leading to several implementations for each story, each with a different set of assumptions and inferences.

The next step of the research process involved looking at the potential applications of narrative analysis to our problem. This provided us with a different approach to performing research, where we focus on computationally analysing a story, instead of primarily focusing on the story itself.

Ultimately, the focus of the software development shifted to the creation of a framework/library for the application of "story thinking" in software engineering. This framework was not planned for initially, rather, it was the spontaneous result of all prior research (especially the worked examples). Here we not only define the theory for many unique and original concepts, but also implement these in code. The result is a Python library which allows for a different approach to software development, and which lessens the gap between how we reason with stories and how we reason with software.

This software report is structured to reflect this development process. First, we look at each of the worked examples and treat each as its own piece of software. We then move on to the narrative analysis component and finish by describing the story-thinking framework.

2.1 Technologies Used

Specific details on any technologies, libraries, or any other relevant items, will be given in each subsequent section, however it may serve well to give a brief outline here of the core technologies used throughout the project. The language of choice for the software development was Python. This was due to personal preference, experience, and also breadth of available libraries and frameworks. Furthermore, Python ties well into another core technology which was heavily utilised – Jupyter Notebooks. Interactive programming was used, first and foremost, as a tool to aid the research; we were studying any potential relationship between literate/interactive programming and our problem. However, it also turned out to be an immensely helpful and hugely powerful tool for quickly testing code and providing examples of code already written. Jupyter Notebooks also allowed for sharing demonstrations easily, documenting thought processes, and directly embedding algorithm visualisations, and even design diagrams. Regarding libraries, there was no single library which was relied on across all software components, however, one worthwhile mention is the well-known graph plotting framework *matplotlib* which proved to be incredibly useful across the board.

2.2 Development Approach to Worked Examples

For brevity, it will be useful here to describe the general approach to working with each example story. As mentioned briefly in the introduction, we start off with a story and transform it before reaching a software implementation, however, it will be beneficial to give more detail on how this process was performed.

Each story that was used as an example comes from some sort of "source" and, therefore, is not ready to be used but instead it is simply a part of a whole, so to speak. Thus, the first step for each example was extracting the actual story itself from the source material. This was easier in some cases than others. It was often found that there was not a clear separation between the story and everything else. Moreover, in many situations, the stories tend to get updated over time. This is especially true for engineering papers where we are provided with the initial story, but this story undergoes several changes as the engineering unfolds. This presented us with major difficulties and care needed to be taken at this step to ensure that we extract the right piece of text. To add to the difficulty, the decision had to be made on whether verbatim versions of the story should be used or not – a verbatim version would be more faithful to the original, however, it would likely be more difficult to work with than a story which had undergo several small "adjustments".

Based on the extracted story, a series of user stories were then drawn up. These lay the foundation of a potential software implementation as they allowed us to see, from a more computational context, what exactly needs to be implemented in code. However, this was also not a walk in the park as the original stories often left a lot of information untold. As such, many inferences and assumptions had to be made at this stage. Additionally, one of the weaknesses (and strengths) of stories is that they can be understood in many different ways. This created further difficulty.

Following from the previous step, a list of assumptions and inferences made was formally defined. This included, not only, the inferences made as part of creating the user stories, but here we also defined any terminology that was not clear in the story and clarified any contradictions. This is a crucial but important step as it not only makes a code implementation more "obvious" but is also a precursor to the next and final stage of the transformation process.

The final step is the design stage. Here we define any design decisions that have been made and class diagrams were drawn up. For more complex implementations, other diagrams were also created such as those visualizing difficult algorithms, or sequence diagrams in the case of threaded implementations.

Once the transformation chain had been completed, a software implementation of story was possible. Note the word "possible" – not "trivial". In many cases, a software implementation was still quite tricky at this stage, simply due to the nature of working with stories. For each of the main worked examples, there can also be said to be another stage which is the "visualisation" stage; that is, we visualise each algorithm/software implementation in an "appropriate" manner suitable to the problem at hand.

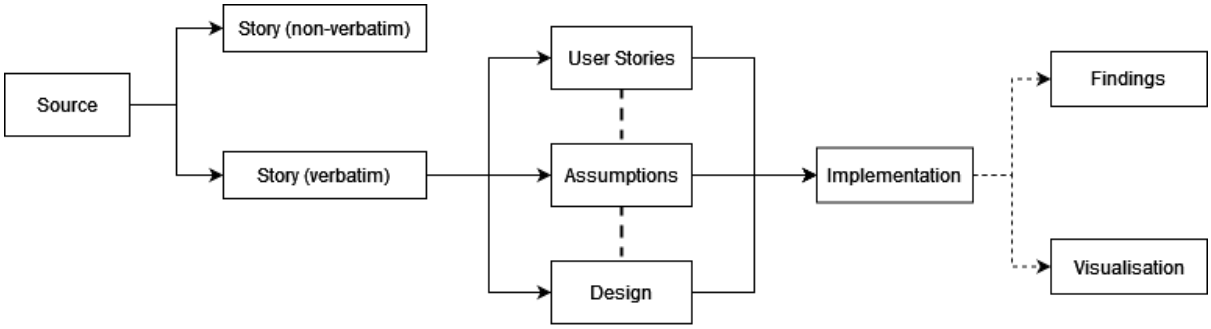

Figure 1: High level overview of worked example transformation chain

3 The Byzantine Generals Problem

The first example looked at is the famous story of the Byzantine Generals (BGP) as specified by Leslie Lamport [1]. The story is used as a means of abstraction to explain the problem of reliable communications across unreliable components.

3.1 Extracted Story

The extracted, verbatim, story used as the software specification for the subsequent software implementation 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. ... They loyal generals should not only reach agreement, but should agree upon a reasonable plan.

3.2 Generals v1

3.2.1 User Stories

As a…	I want to…	So that…
General	Make observations on an enemy city	We (generals) can decide on a plan of action
''	Be able to communicate with other generals	We can share our observations
Loyal General	Decide upon a reasonable plan of action	My division does what is most reasonable
''	Decide upon the same plan of action as all other loyal generals	Order within the army is maintained
Disloyal General	Make false/contradicting observations	The loyal generals do not reach agreement
Messenger	Deliver messages between generals/divisions	The generals can reach upon some agreement

Table 1: User stories for BGP

3.2.2 Assumptions

- There is only one messenger.
- "Enemy" and "Enemy City" are taken to refer to the same idea.
- A "reasonable plan" is assumed to mean that which wins a majority vote.
- The observations that generals can make are either "attack" or "retreat".

3.2.3 Design Decisions

- The enemy/city is taken to have no functionality other than just "existing".
    - A city is assumed to be objectively either "attackable" or not, i.e., all generals make the same observations on the same city.
- A traitorous general simply "lies" to the other generals, i.e., he communicates the opposite of his observations to all generals.
- Enemy cities are instances of a class that does nothing.
- Each general works by first making an "objective" decision on an enemy. This is his true observation and not necessarily the one he communicates to the others.
- There is no tie breaker for the majority vote.

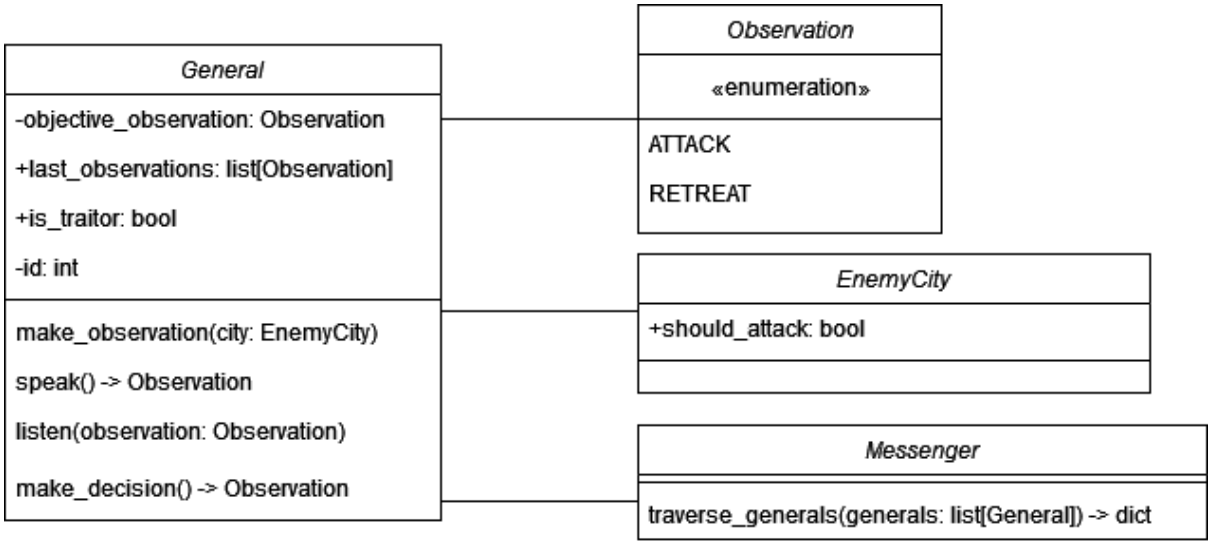

Figure 2: Generals v1 system design

3.2.4 System Overview

For this worked example (this version and all subsequent versions) we focus specifically on the communication algorithm used by the generals to reach agreement. We model each general and messenger, and enemy city as an instance of a class. The structure of the *main* method is roughly as follows.

- Initialise instances of loyal generals and traitorous generals. The specific numbers of each can be customised with a set of parameters by passing either absolute values, percentages, or indicating that a random allotment should be used.
- Initialise enemy city instance. The value of *should_attack* is parameterised and can be customised.
- Each general makes an "objective" observation on the city.
- The messenger traverses all generals and generals share their observations between each other. The algorithm used for this is simple; each general simply reports his observation to every other general. Loyal generals report their objective observation, traitorous generals report the opposite of their objective observation.
- Each general comes up with a plan of action based on the collected observations. This is simply a majority vote.
- Appropriate output is printed.

3.2.5 Testing

In addition to the main method being highly customisable via flexible parameters, a simple test runner was implemented which accepts a series of "test cases" and runs each in turn. A test case is defined by the total number of generals, and the number of traitorous generals. This test runner additionally is able to re-run each test multiple times. This is especially useful for the future versions as they have non-deterministic results. The output of each test is collated and reported in an easy-to-understand textual format. Various test cases were performed as part of development to ensure the correct operation of the system. See the below figure for sample output (running with 10 total generals, 4 of whom are traitorous, and an "objectively attackable" enemy city).

```
Running test (10, 2)                    Running test (10, 4)                    Running test (10, 6)
--------------------                    --------------------                    --------------------

[PLAN=Observation.ATTACK ATTACK=72 RETREAT=18] [PLAN=Observation.ATTACK ATTACK=54 RETREAT=36] [PLAN=Observation.RETREAT ATTACK=36 RETREAT=54]

<General 31 [l]>: attack=7 retreat=2     <General 61 [l]>: attack=5 retreat=4     <General 91 [l]>: attack=3 retreat=6
<General 30 [l]>: attack=7 retreat=2     <General 60 [l]>: attack=5 retreat=4     <General 90 [l]>: attack=3 retreat=6
<General 32 [l]>: attack=7 retreat=2     <General 62 [l]>: attack=5 retreat=4     <General 92 [l]>: attack=3 retreat=6
<General 33 [l]>: attack=7 retreat=2     <General 63 [l]>: attack=5 retreat=4     <General 93 [l]>: attack=3 retreat=6
<General 34 [l]>: attack=7 retreat=2     <General 64 [l]>: attack=5 retreat=4     <General 94 [t]>: attack=4 retreat=5
<General 35 [l]>: attack=7 retreat=2     <General 65 [l]>: attack=5 retreat=4     <General 95 [t]>: attack=4 retreat=5
<General 36 [l]>: attack=7 retreat=2     <General 66 [t]>: attack=6 retreat=3     <General 96 [t]>: attack=4 retreat=5
<General 37 [l]>: attack=7 retreat=2     <General 67 [t]>: attack=6 retreat=3     <General 97 [t]>: attack=4 retreat=5
<General 38 [t]>: attack=8 retreat=1     <General 68 [t]>: attack=6 retreat=3     <General 98 [t]>: attack=4 retreat=5
<General 39 [t]>: attack=8 retreat=1     <General 69 [t]>: attack=6 retreat=3     <General 99 [t]>: attack=4 retreat=5
```

Figure 3: Generals v1 example output

3.2.6 Observations

In this implementation, the loyal generals **always** reach agreement. Sometimes they agree on an "unreasonable" plan. This behaviour is, effectively, the result of a number of factors, the biggest being the consistency that is introduced as a result of every loyal general making the same observation and every traitorous general simply reporting the opposite. Thus, so long as the number of loyal generals is greater than the number of traitors, the loyal generals will agree to the same plan. If, however, there is a tie, the first observation gets returned. As the observations are stored in lists (ordered) and the generals are iterated in order (starting with the loyal generals), the first observation in the list is always that of a loyal general. Note that "tie" in this situation refers to a tie in the number of observations, i.e., the same amount of ATTACK observations as RETREAT. As for a tie in terms of the numbers of generals, the loyal generals always adopt the plan of the traitorous generals. This is because for *n* loyal generals, each *loyal* general has *n-1* true observations and *n* false observations. This is because a general does not consider his own observation when making his decision. If there are more traitors than loyal generals, the majority wins and the loyal generals agree upon an unreasonable plan. Moreover, this representation of the story is unlikely to be an appropriate one due to the simplicity of the communication algorithm. As mentioned before, each general simply communicates his observation to every other general. Therefore, the logical conclusion (which can be verified by Fig. 3) is that if we have more traitors, they will win over and vice-versa. This is not an adequate representation of the problem described by Lamport.

3.3 Generals v1a

This version is a minor modification of *v1* where the traitors, instead of lying, return random observations. In this version, the majority still wins, so the behaviour is unchanged where there is no ambiguity in this regard. Likewise, if there is a tie in the number of observations, the behaviour is the same as before. However, if there is a tie in the number of generals, it is now possible for the loyal generals to fail to reach agreement. It is also far less likely for the loyal generals to adopt an unreasonable plan now. This makes sense if we assume that the "random" split between the observations that the traitorous generals make is 50/50; there is no impact on the majority. In Fig. 4 we can see the non-deterministic behaviour of this implementation (10 generals, 6 traitors).

```
[PLAN=Observation.ATTACK ATTACK=62 RETREAT=28] [PLAN=Observation.ATTACK ATTACK=64 RETREAT=26] [PLAN=None ATTACK=65 RETREAT=25]

<General 91 [l]>: attack=5 retreat=4          <General 101 [l]>: attack=6 retreat=3         <General 111 [l]>: attack=5 retreat=4
<General 90 [l]>: attack=5 retreat=4          <General 100 [l]>: attack=8 retreat=1         <General 110 [l]>: attack=4 retreat=5
<General 92 [l]>: attack=7 retreat=2          <General 102 [l]>: attack=6 retreat=3         <General 112 [l]>: attack=7 retreat=2
<General 93 [l]>: attack=7 retreat=2          <General 103 [l]>: attack=7 retreat=2         <General 113 [l]>: attack=6 retreat=3
<General 94 [t]>: attack=7 retreat=2          <General 104 [t]>: attack=6 retreat=3         <General 114 [t]>: attack=8 retreat=1
<General 95 [t]>: attack=6 retreat=3          <General 105 [t]>: attack=6 retreat=3         <General 115 [t]>: attack=7 retreat=2
<General 96 [t]>: attack=5 retreat=4          <General 106 [t]>: attack=6 retreat=3         <General 116 [t]>: attack=6 retreat=3
<General 97 [t]>: attack=7 retreat=2          <General 107 [t]>: attack=7 retreat=2         <General 117 [t]>: attack=9 retreat=0
<General 98 [t]>: attack=6 retreat=3          <General 108 [t]>: attack=5 retreat=4         <General 118 [t]>: attack=6 retreat=3
<General 99 [t]>: attack=7 retreat=2          <General 109 [t]>: attack=7 retreat=2         <General 119 [t]>: attack=7 retreat=2
```

Figure 4: Generals v1a example output

3.4 Generals v2

This implementation follows from *v1a* but makes one small (but significant) change in that each general now makes different observations on the same city. This is perhaps a more realistic approach, as in a real situation, if generals were camped outside an enemy city, it is likely that some would want to attack while others would prefer to retreat.

3.4.1 Changes in This Version

- Whether a city is "attackable" or not is now dependent on each general, i.e., *should_attack* is no longer a static Boolean property.
- Generals make "random" initial (objective) observations on a city.
- Traitorous generals randomise the decision that they communicate to others.
- Tie breaker is not strictly necessary anymore due to added randomness.
- As an interesting "side-effect" the *EnemyCity* class is now entirely useless as the logic inside *EnemyCity.should_attack* could be moved into *General.make_observation*.

3.4.2 Design

The design of the system is the same as *v1* (Fig. 2) with the only exception being the *EnemyCity* class, which now has the following signature (Fig. 5).

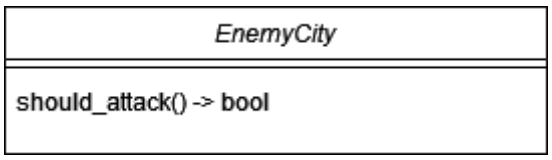

Figure 5: Generals v2 design update

3.4.3 Observations

As before, the majority vote wins so in cases where one observation is more popular than the other, the behaviour is unchanged. However, the problem we now have is that the loyal generals very often fail to reach an agreement due to the removed consistency and added randomness. Each loyal general makes a different observation and each traitor reports a different observation to each other general. Thus, getting all the loyal generals to agree to the same plan is an exceedingly rare occurrence. Therefore, although the behaviour of this implementation might be the same as before if the state of the "simulation" matches, achieving that same state, even with the same parameters, is very difficult.

Another thing to note is that in this implementation the line between a loyal general and a traitorous one has become somewhat blurred in that both make initially random observations. The loyal generals simply return these first observations whereas the traitors randomize again before returning. Conceptually this mimics the idea of the traitors simply lying about their initial observation, but logically, there is no difference. The added randomness changes nothing. This is different to the initial implementation where a city was "objectively" attackable or not. It seems the only difference here

between the traitors and the loyal generals is that the loyal generals report the same observation to every general, whereas the traitors may not.

Following from this, it is interesting to note how by removing the concept of an objectively attackable city, the definition of a "false" and "true" plan (as talked about in the previous versions) becomes ambiguous. We have made the assumption that a "reasonable" plan is that which wins a majority vote, however, if every observation made is totally random and independent of a city, the meaning of a majority vote gets somewhat lost.

3.5 Generals v2a

In an attempt to "fix" some of the issues with *v2,* this implementation simply keeps the initial randomness of the observations but reverts to the behaviours of the traitors from *v1* where they simply "lied". This added consistency (or "reduced randomness") makes it far less likely for the loyal generals to *not* reach agreement. This, however, means that the impact of the traitors is negligible if we assume that the split for observations is effectively 50/50.

3.6 Generals v3

This is a major revision of the communication algorithm employed by the generals. Here we utilise the recursive *OM* algorithm as described by Lamport in the original paper.

3.6.1 Design

- Recursive OM algorithm.
- Loyal generals return the order that the last commander made.
- Traitors return random orders.

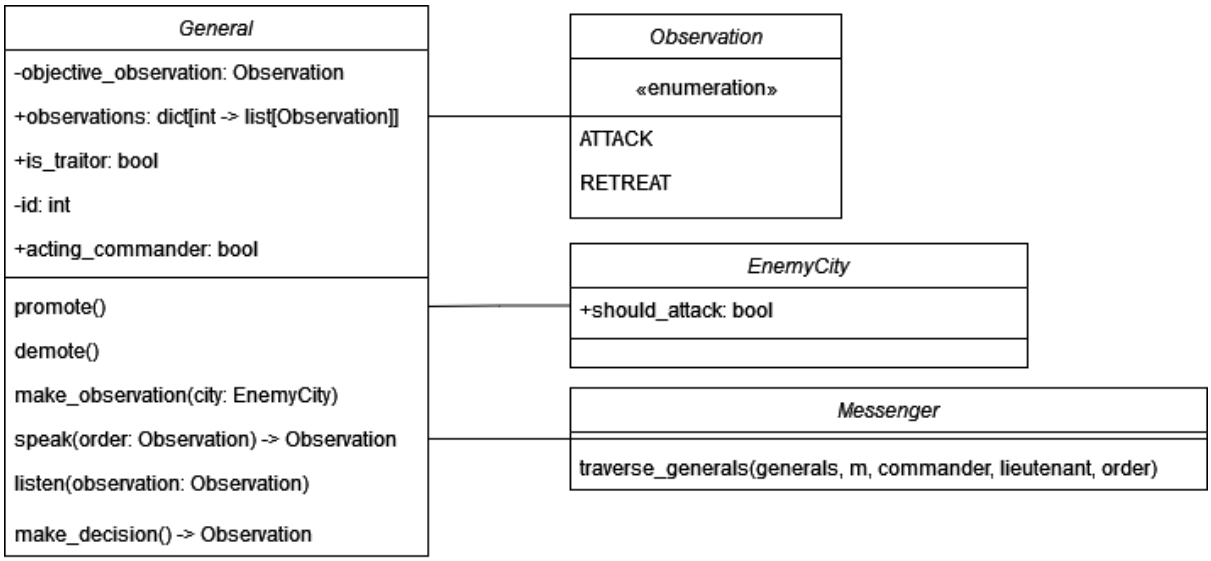

Figure 6: Generals v3 design

3.6.2 The Communication Algorithm

The *OM* algorithm as described by Leslie Lamport is shown in the below figure.

Algorithm OM(0).

(1) The commander sends his value to every lieutenant.
(2) Each lieutenant uses the value he receives from the commander, or uses the value RETREAT if he receives no value.

Algorithm OM(m), $m > 0$.

(1) The commander sends his value to every lieutenant.
(2) For each i, let v_i be the value Lieutenant i receives from the commander, or else be RETREAT if he receives no value. Lieutenant i acts as the commander in Algorithm OM($m - 1$) to send the value v_i to each of the $n - 2$ other lieutenants.
(3) For each i, and each $j \neq i$, let v_j be the value Lieutenant i received from Lieutenant j in step (2) (using Algorithm OM($m - 1$)), or else RETREAT if he received no such value. Lieutenant i uses the value *majority*($v_1, \ldots, v_{n-1}$).

Figure 7: OM algorithm specification

The pseudo code implementation of the above algorithm is shown in the following listing.

```
OM(generals, m, commander, lieutenant, order)

    if m < 0
        if commander is not lieutenant
            lieutenant.listen(order)
    elif m == 0
        for g in generals
            OM(generals, m-1, commander, lieutenant, commander.speak(order))
    else
        for g in generals
            if not g.is_commander
                g.is_commander = True
                OM(generals, m-1, g, None, g.speak(order))
                g.is_commander = False
```

One of the most apparent changes in this implementation is the introduction of the concept of a commanding general. Before the first execution of the OM algorithm, a general is picked to be a commander at random, and the commander makes his observation on the enemy city. The OM algorithm then runs with this general as the initial commander. For each recursive step in the algorithm, every other lieutenant acts as the commander and reports their observation to the others. Note that each commander reports the order that the previous commander made, not the observation that *they* made. Consider the following example with 4 generals (1 traitor) and two steps in the algorithm.

Commander says ATTACK to generals 1, 2, 3.

> *General 1 says the commander said ATTACK to generals 2, 3.*
>
> *General 2 says the commander said RETREAT to generals 1, 3.*
>
> *General 3 says the commander said ATTACK to generals 1, 2.*

3.6.3 Comments and Observations

This implementation was, surprisingly, more challenging than anticipated. This mostly stemmed from various contradictions and confusing terminology in the original BGP paper. For starters, the introduction of the concept of a commanding general was a matter of some confusion as this does not exist in the original story – it gets "appended" later on as part of the engineering in the paper. Furthermore, there exist some contradictions regarding the functioning of the algorithm itself, not only in online sources, but also in the original paper itself. To try to address some of these issues, various different implementations of the OM algorithm were experimented with, however, none proved entirely satisfying in that none of them verified the claim that Lamport makes of needing *3m+1* generals to cope with *m* traitors. Additionally, we compared our implementation with various other implementations online and it was discovered that all suffered from the same problem. We found that most of these implementations assumed that traitors simply return the opposite observation some percentage of the time, however if we made a minor modification that made the traitors return the opposite observation 100% of the time, the claim of *3m+1* generals always failed. One such implementation is that provided by JVerewolf on Github [2]. Interestingly, his implementation is largely similar to ours with one major difference – the only class in this implementation is that of the *General*. All the communication in JVerewolf's implementation happens within this class. This, we believe, is a superior design as it is far simpler and more elegant than trying to work with a *Messenger* class. Having a *Messenger* class, however, results in an implementation which is more faithful to the original story.

3.6.4 Visualisation

As part of this implementation, a simple method for visualising the problem was written with the help of the *matplotlib* library. Initially, an attempt was made to draw a node graph, however this proved to get very unreadable with increasing numbers of generals, therefore, a simple bar graph was settled for in the end.

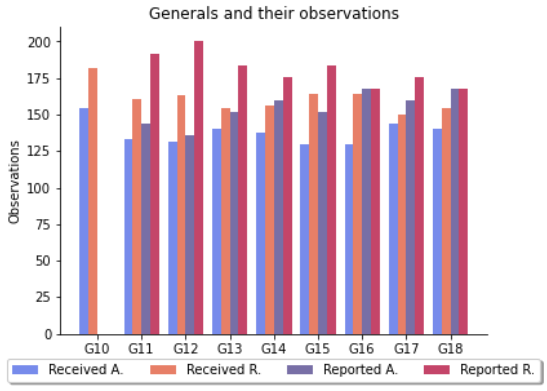

Figure 9: 3m generals

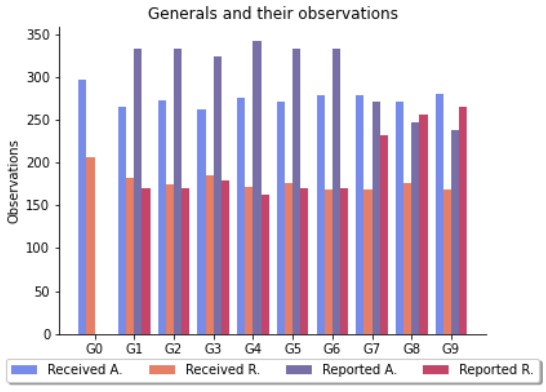

Figure 9: 3m+1 generals

3.7 Generals v4

In an attempt to investigate the various methods of representing "agency" in software (i.e., ways of mapping characters into agents that perform actions), we take a threaded approach to the final implementation of the BGP.

3.7.1 Design Decisions

- Traitors always return retreat.
- Generals and Messengers run on their own threads. This is implemented with the built-in Python library *threading*.
- The code is modelled to be thematically accurate, thus:
    - Each general has "his own" messenger.

    - When a general “speaks”, he dispatches jobs to his messenger who then delivers them to other generals.
    - When a general “listens”, he adds what he heard to his list of observations and dispatches his messenger to propagate the message.
    - The commander kick-starts the cycle.
- The recursive algorithm is replaced by threads executing jobs placed on a queue.
    - This makes it difficult to know when we’re finished – a timeout of 2 seconds is used for this purpose.
- There is no sophisticated integrity/authenticity checking of any kind, nor does this code verify the claims made in the BGP paper. It is merely a proof of concept for demonstration purposes.

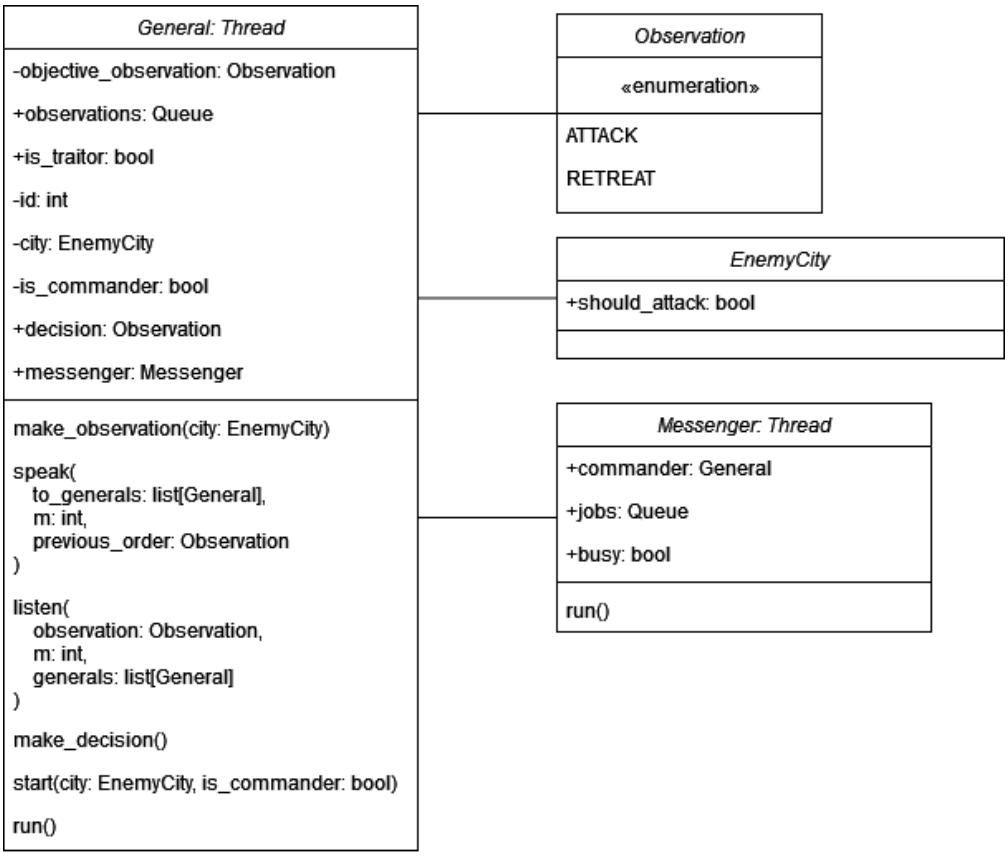

Figure 10: Generals v4 design

3.7.2 Observations and Comments

This implementation is interesting as it is likely the most complex and also the most thematically accurate. The algorithm used is a slightly modified version of the recursive OM algorithm, where the recursion is replaced by threads executing jobs on a queue. If we consider, at a high level, what is happening we see that it maps very nicely to the original story:

- Each general “exists” simultaneously (due to threading).
- The commander dispatches his messenger with an order which is to be delivered to the other generals.
- Each general, in turn, then dispatches *his own* messenger to deliver *his* order to every other general.
- The generals then make some sort of decision based on all received orders.

To better understand the algorithm and flow of execution, consider the following diagram where *C* represents the commander, *L* the loyal general, and *T1* and *T2* two traitorous generals. Messengers have been omitted for clarity. Moreover, since the code is threaded, it can be assumed that the order of communications (represented by edges on the graph) is, effectively, simultaneous.

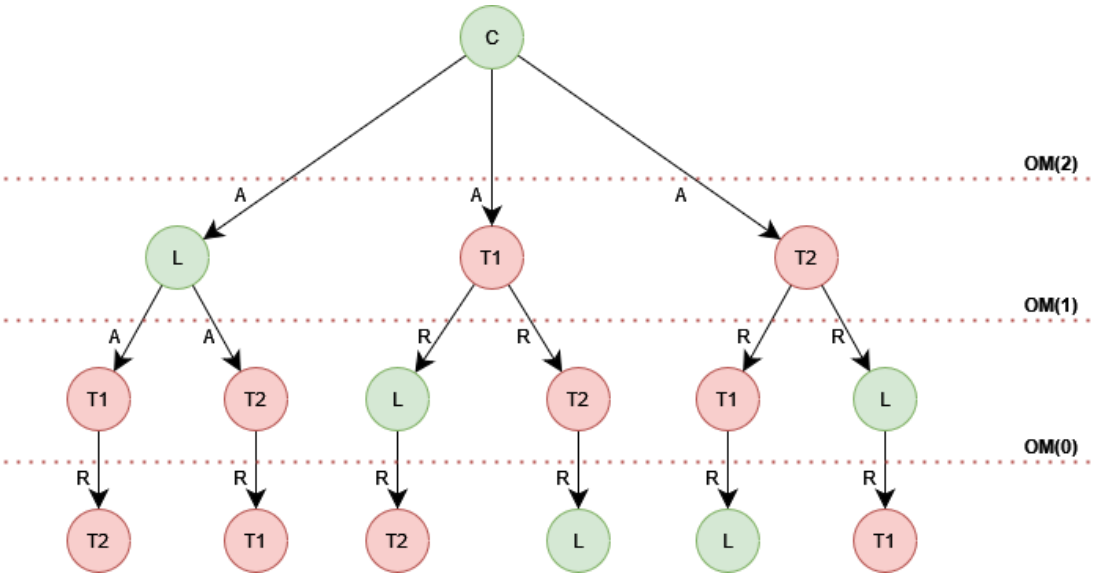

Figure 11: BGP algorithm visualisation

Let us also consider a sequence diagram showing the interactions between the threads.

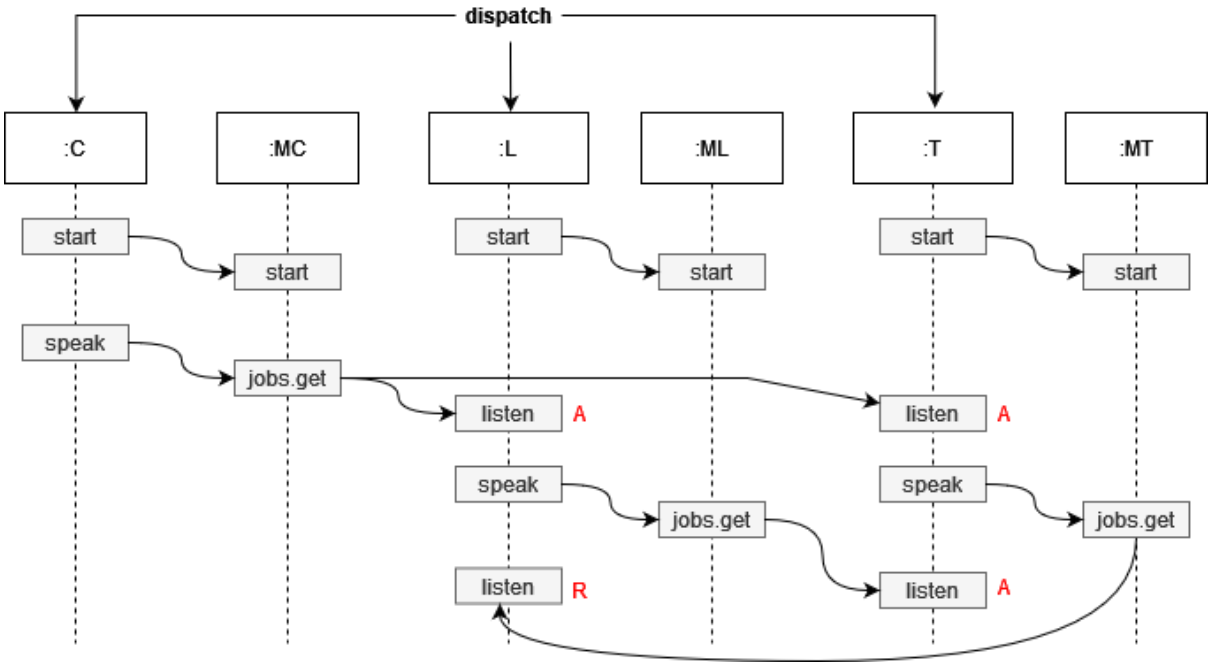

Figure 12: Generals v4 sequence diagram

4 The Dining Philosophers

We now look at another famous engineering problem known as the Dining Philosophers, as given to us by Edsger Dijkstra and Tony Hoare [3].

4.1 Extracted Story

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 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.

4.2 Philosophers v1

4.2.1 User Stories

As a…	I want to…	So that…
Philosopher	Think	I can ask questions and get answers
"	Eat	I can continue thinking
"	Have access to two forks	I can continue eating

Table 2: User stories for the dining philosophers

4.2.2 Assumptions

- At any given moment, a philosopher is either thinking or eating. If the philosopher is not eating, it can be assumed that he is thinking and vice-versa.

4.2.3 Design Decisions

- Philosophers think by default.
- Philosophers think until they start eating and eat until they start thinking.
- Forks are modelled as classes so they can be passed by reference to each philosopher.

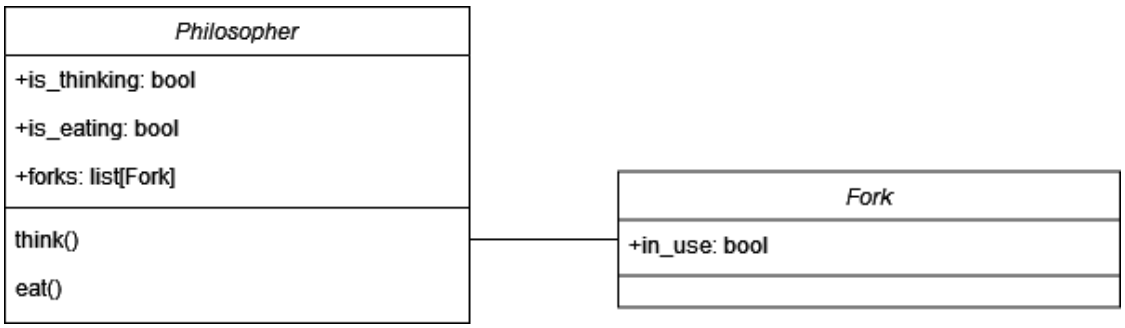

Figure 13: Dining philosophers design specification

4.2.4 System Overview

For this story, the software implementation is somewhat simpler than that BGP implementation. Here, we simply model each philosopher and fork as a class. We also define a custom error class *ForkInUse* which gets triggered when an attempt is made to eat with a fork which is already being used. We initialise the philosophers and forks, and then (manually) try to get each philosopher to eat. Consider the following pseudo-code:

> Philosophers[0].eat()

> Philosophers[1].eat()

Error: ForkInUse

This is purposefully a simplistic design as this initial implementation serves only as a demonstration of the problem. We expand on this in the subsequent versions.

4.3 Philosophers v2

In this implementation we build on the previous version by modelling each philosopher as a thread. Other than the added threading functionality, the design is largely the same as before. This version is, of course, a much more appropriate model of the original story as the original problem is one of deadlock during threading. This version, however, is still just a "model" of the problem – it does not solve it. Though, this time we do not have to "manually" call any methods, we simply start all the threads and the philosophers kick off their cycle of eating and thinking. As before, the *ForkInUse* exception gets raised due to multiple philosophers trying to eat with the same fork. Further details and the design diagram are omitted for brevity as this gets revisited and built upon in the following section.

4.4 Philosophers v2a

In this final implementation, we expand on *v2* by implementing the deadlock solution as described in the original research paper (Fig. 14).

```
In this universe the life of philosopher w can now be coded
cycle begin think;
            P(mutex);
                 C[w]:= 1; test (w);
            V(mutex);
            P(prisem [w]); eat
            P(mutex);
                 C[w]:= 0; test [(w+1) mod 5]; test [(w-1) mod 5];
            V(mutex)
      end
```

Figure 14: Dining Philosophers algorithm (EWD310, page 23)

4.4.1 Design and Overview

- Each philosopher is a thread.
- Each philosopher has a "cycle duration" which dictates for how long the philosopher thinks and eats. This is for demonstration purposes.
- Philosophers alternate between thinking and eating for as long as the thread is not told to stop.
- Each philosopher emits events when he is hungry, thinking, and eating. These get collected in a queue. This is for visualisation purposes.
- The philosopher *class* has a *table* attribute (note: class, not instance) which gives access to every philosopher *instance*. This is necessary to implement the deadlock solution.

- A *test* method is implemented as described in the original paper. The code for this is given below.

```
def test(k):
    c = Philosopher.table

    if c[(k-1) % len(c)].c != 2 and c[k].c == 1 and c[(k+1) % len(c)].c != 2:
        c[k].c = 2
        c[k].prisem.release()
```

To describe the algorithm simply:

- Philosopher threads get started.
- Philosopher thinks.
- Philosopher indicates that he is hungry and attempts to eat.
- The philosopher eats for a certain duration when this becomes possible.
- Philosopher indicates that he has finished eating.
- Philosopher thinks again.

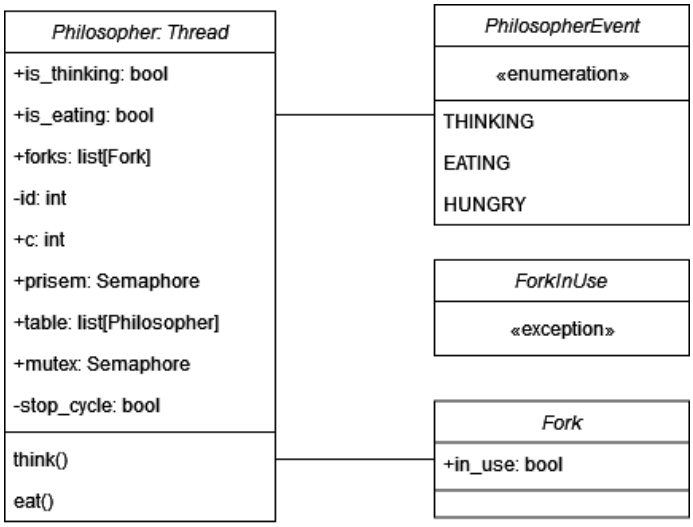

Figure 15: Dining Philosophers v2a design

4.4.2 Visualisation

As mentioned earlier, we use the events queue to draw a graph using the *matplotlib* library. In this situation, a timeline of philosopher events is a perfect choice. See the below figure.

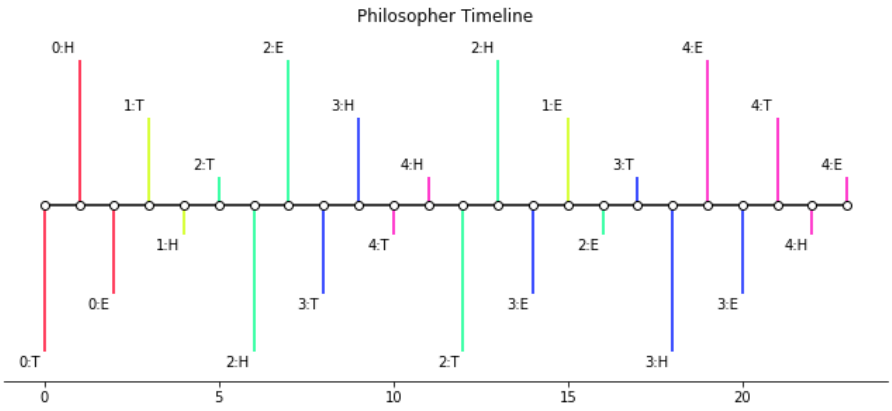

Figure 16: Dining Philosophers visualisation

From the above diagram, we can see that we have successfully modelled the problem described in the story and solved it adequately. Every philosopher is able to think and eat, with no exceptions being thrown and with no deadlock.

4.5 Testing

There was no need for any extensive testing with the Dining Philosophers implementations as it was pretty clear when the code did what it needed to do. For the first two implementations, we were simply interested in reaching our error condition of two forks being accessed simultaneously. The idea of this was to demonstrate the problem as described in the original paper. This was easy to do by simply running the code until an exception was thrown. Testing the final implementation, however, was not as simple as this due to the (now correctly running) concurrency. This is where the timeline visualisation and philosopher events come in. By simply inspecting the generated timeline, we can see that the code does what it should, and it works correctly without throwing any exceptions – each philosopher thinks, gets hungry, and eats with no issues.

5 Hansel and Gretel

The final worked example is that of the famous German fairy tale “Hansel and Gretel”. This is an interesting example as it is completely different to the two previous examples looked at. In the previous examples, we were working with stories which were used to abstract away engineering problems; here however, the whole text is a story in its own right. Moreover, it is a story which has no connection whatsoever to computing.

5.1 Extracted Story

Though extracting the “story” was quite a challenge (because it was all a story), the excerpt we decided to work with in the end is as follows.

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 may be difficult to imagine what piece of software can be written based on the above excerpt, thus we should note here that in our implementation of Hansel and Gretel, we focus on the “path-finding” algorithm used by Hansel to find his way back home.

5.2 Hansel v1

5.2.1 User Stories

As a…	I want to…	So that…
Hansel	Drop pebbles	I can mark my path from home
"	Follow my dropped pebbles	I can get back home

Table 3: Hansel and Gretel user stories

5.2.2 Assumptions

- Pebbles are not necessarily dropped in a straight line.
- Hansel keeps track of which pebbles he has already visited.

5.2.3 Design Decisions

- Hansel’s path is modelled as a 2D graph where each pebble dropped has its co-ordinate in this two-dimensional space.
- A “random walk” is generated to simulate Hansel’s path. This works by simply generating random points within a radius from an origin, provided that these points Hansel closer to the destination.
- Hansel works his way back based on which pebbles are visible from his current location. This may be one pebble, or it may be more. The path he picks is not necessarily the same one whith lead him to the destination. He may visit less pebbles, or he may get stuck by visiting the “wrong” ones.

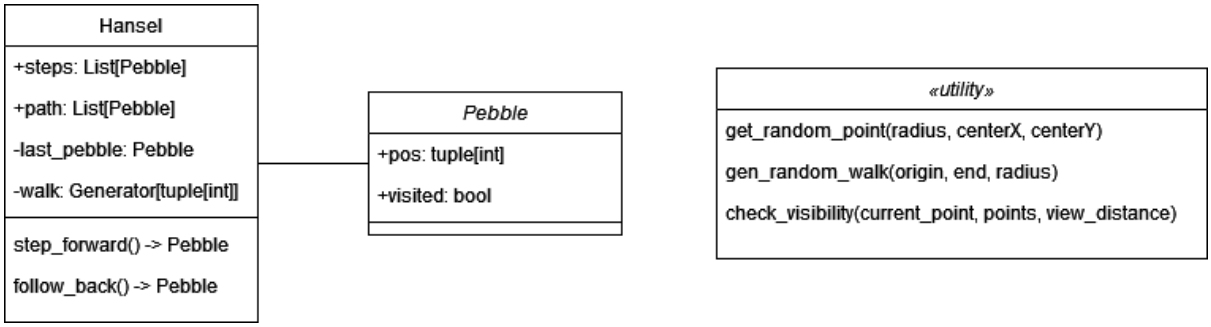

Figure 17: Hansel and Gretel design

5.2.4 System Overview

In this implementation, we not only model the story itself, but also "solve" it by implementing the path finding algorithm used by Hansel. We model Hansel as an instance of a class and "pre-generate" a random walk [4] when initialising this class. This is done with the *gen_random_walk* class which generates a list of "random" points starting from *origin* and ending at *dest*. Each point is within a specified *radius* of the last point, and points are only accepted if they get Hansel closer to *dest*. For demonstration purposes, these values are defined as constants. In terms of the story, the *origin* is Hansel's house, *dest* is the point in the forest at which Hansel decides to turn back, and *radius* is how often Hansel drops a pebble. Using the Hansel instance, we are able to step through this random walk and simulate Hansel walking into the forest. When following back, Hansel considers all pebbles which are visible from his current point and moves to one which he hasn't visited before. This way, he is able to work his way back until he reaches his starting position. This is the algorithm implemented in the method *Hansel.follow_back*.

5.3 Hansel v2

The second implementation is, perhaps, not necessarily an alternate implementation in its own right, but rather an "extension" of the previous version. Here we simply attempt to build a more generic and abstract version of the previous code. We do this by extending on *v1* and implementing a single method which is defined as *walk_graph(nodes, edges, start, end).* This function implements a simple algorithm for traversing pebbles just as Hansel did, however, it does this by traversing a generic graph described by a set of nodes and edges. Each node represents a pebble, and an edge represents visibility between pebbles. This implementation actually proved quite useful for comparison purposes with the original implementation as it highlights that small differences in the logic of a software implementation can lead to significant changes in behaviour relative to the original story, as we will see in the following section.

5.4 Visualisation

As usual, we develop a simple method with the help of *matplotlib* for visualising the implemented algorithms. Here, quite logically, we settle for a 2D plot of the steps that Hansel makes. The blue and red points represent all pebbles that Hansel dropped on his way into the forest; the red points mark the path he took on his way back.

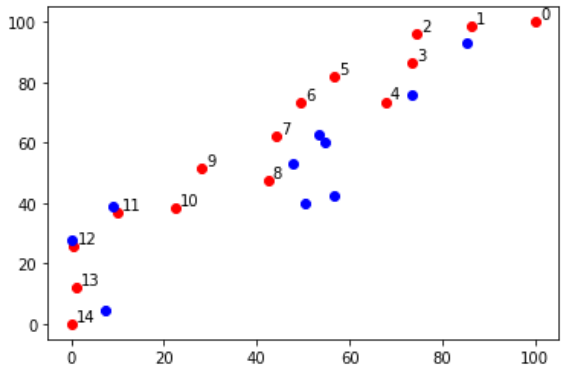

Figure 19: Hansel v1

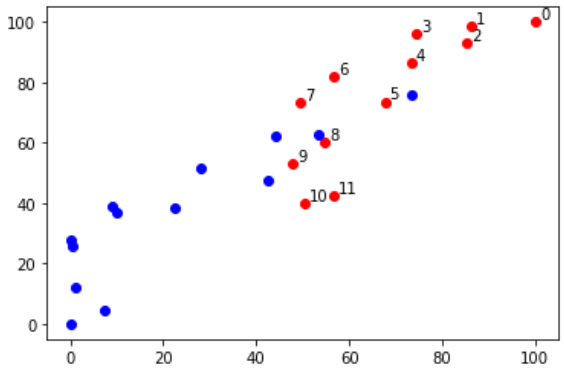

Figure 19: Hansel v2

Here we are able to observe the differences mentioned at the end of Section 5.3. Notice how in the second implementation, Hansel actually gets stuck and is unable to return back home, while in the first implementation, he finds his way back home with no issues. This is due to the order in which unvisited pebbles are iterated. This small difference between implementations results in significantly different behaviour and highlights the challenge in effectively mapping a story to a software implementation.

5.5 Testing

The success conditions for each implementation are Hansel either finding his way back to the *origin* point or running out of possible moves (without entering an infinite loop). Both of these conditions were trivial to verify manually and to ensure that the software is working as expected. Moreover, the visualisation as shown in the previous section provides another mechanism by which we can verify that the software is running without issues. If there were any problems with the code, we would see that reflected in the visualisation (or lack of).

6 Narrative Analysis

We now take a slightly different approach and shift our focus to investigating how we can programmatically analyse a story. The result of this approach is a series of Python scripts and utility functions to perform narrative analysis on a given input. Two primary approaches are used here:

1. Named entity recognition (NER) with the help of the *flair* [5] Python library.
2. Comprehensive narrative analysis using *spaCy* [6] and *Narcy* [7].

See the below figure for an overview of the narrative analysis components.

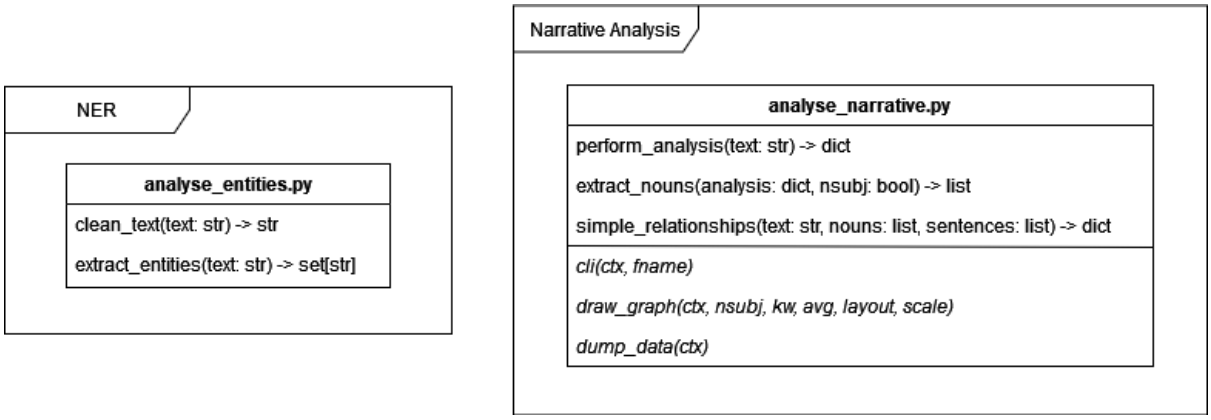

Figure 20: Narrative analysis design overview

6.1 Named Entity Recognition

This was the first approach to narrative analysis and is a very minimal and simple software component. We simply define a function which cleans up any text and another function which takes this input and attempts to extract named entities. This is done with the *flair* library and the built-in tagger that this provides. No further work was done on this front simply due to the unsatisfying results of this approach. The software was tested on the three main stories looked at in the previous sections, unfortunately this proved to be somewhat disappointing. For example, when running with the BGP story, an ideal output would be *{General, Messenger}*, however, in practice, an empty set gets returned.

6.2 Further Narrative Analysis

Following from the above, a more comprehensive approach was employed where, instead of simply relying on NER, narrative analysis was employed with the help of Narcy and spaCy. The results of this are far more promising. Internally, this works by utilizing the aforementioned libraries to perform supervised narrative analysis with pre-trained models provided by spaCy. Once a piece of text is fed into the program, it gets cleaned up, parsed, and analysed. The analysis step is able to (currently) extract sentences, tokens, token relationships, part of speech tags, token dependencies, sentiment scores, etc...

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 somewhat acceptable.

Once the nouns are extracted, the relationships between each noun are established. This is implemented with a crude algorithm which simply determines which nouns occur together in a sentence. This is certainly an area of potential improvement as there are likely to be better and more fine-grained alternatives to this approach, however, the results of this are (surprisingly) quite good.

The output produced by this script is a node graph which visualises each noun and the relationships between each noun. This is done with the help of the *NetworkX* [8] library. This functionality is exposed

via a command line interface using the *click* [9] Python library. Using this interface, it is possible to pass various "filters" to the analysis/extraction process. For example, we can limit the nouns to contain only nominal subjects, filter nouns only to those with an above average occurrence, and constrain nouns only to those which are among the top X keywords (keyword extraction was performed with the *yake* [10] library). Furthermore, options to customise the output graph are also available, such as configuring the node-positioning algorithm and adjusting the scale of the output.

This script was tested with each of the primary stories to ensure it functions correctly. See the below figures for examples of what the output may look like.

MAYBE WE CAN DEVELOP SOME SIMPLE UNIT TESTS FOR THIS

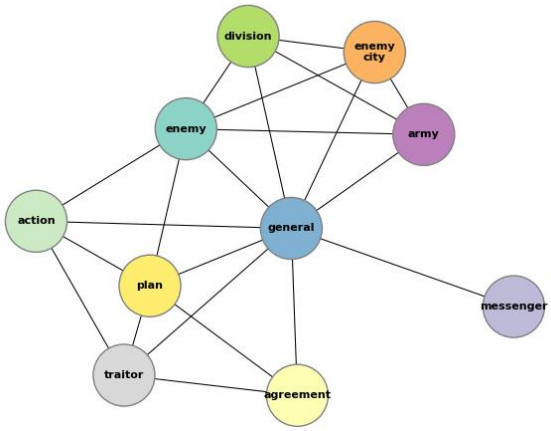

Figure 22: BGP narrative analysis

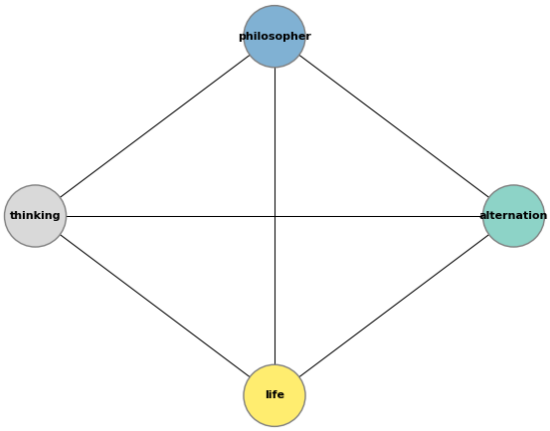

Figure 22: Dining philosophers narrative analysis

7 Framework For Story-thinking in Software Engineering

The culmination of all work discussed thus far is a Python framework/library for the application of story-thinking in software. What we mean when we say "framework for story-thinking in software engineering" is that this is a Python library which allows us to reason about software more similarly to how we would reason about stories. See the below diagram for a high-level overview of the framework and its components.

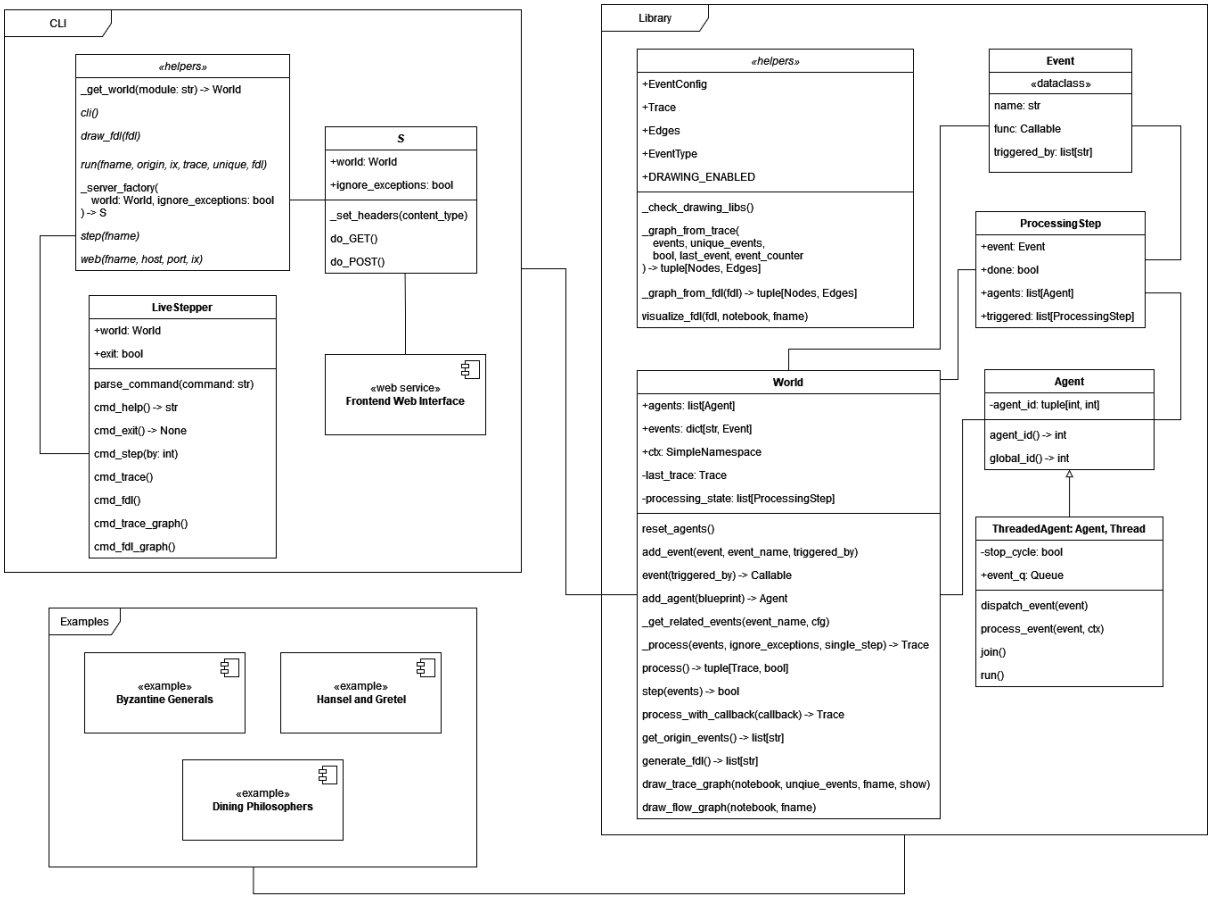

Figure 23: Story-thinking framework design overview

7.1 Story-thinking in Software Engineering

Upon carefully analysing the code written for the three primary stories looked at, it was discovered that if we break down the story and the software into a set of "basic" elements, there is a lot of overlap between the elements of the story and the elements of its software implementation. Briefly, the software elements that were decided on are as follows:

- Agents.
- Events.
- Actions.
- Components.
- Flow.

More detail on these can be found in the companion research article. Thus, the proposed Python library allows us to efficiently and effectively model software in terms of these elements.

When using the framework/library, we structure code as having a global *World* instance; each *World* instance is composed of *Agents* and each *Agent* can perform *actions*; the *World* itself can also emit

events. Each event and action can trigger further events and actions – this is how the software performs its logic. This is exposed via a user-friendly interface where agents are defined by simple subclassing either the *Agent* or *ThreadedAgent* class. Events and actions are defined by using the *event* decorator which is bound to every world instance. For example, consider the following code snippet:

```
w = World()


class Foo(Agent):
    @w.event('baz')
    def bar(self, ctx):
        ...


@w.event()
def baz(ctx):
    w.add_agent(Foo)


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

In the above example, we define an agent *Foo*, which performs a single action *bar*. This action gets automatically triggered when the global world event *baz* is triggered. When we call *w.process*, the world instance automatically figures out that *baz* is the top-level event and calls it, adding the agent and, ultimately, triggering the action *Foo.bar*. More complex event interactions are also supported by returning a special value from each event (*EventConfig*); this allows overriding the default event/action interaction. Support for threaded agents is also available by simply subclassing the *ThreadedAgent* class – no extra work is necessary; all the heavy lifting is done by the library itself. The processing of a world results in what we refer to as the *trace*. This is simply a dictionary which maps all the interactions that took place during the runtime of the world. For the above example, the trace may look like the following:

```
[
    {
        "event": "baz",
        "triggered": [{"event": "Foo.bar", triggered: [], "agent": (0, 0)}]
    }
]
```

The above trace provides a very powerful mechanism for visualising complex algorithms by allowing us to draw a node graph of the interactions between each event and agent action. This funciontality is achieved by using the *NetworkX* and *pyvis* [11] libraries. These are the only non-standard libraries used

as part of this framework – all other functionality is either coded from scratch, or with the help of the Python standard library. There are two options for drawing the graphs – one where we treat *every* instance of the same action/event as a single node, and the other where we visualise each individual action/event as its own, unique node.

One of the most powerful functionalities is the ability to "live-step" through the world – that is, rather than processing everything at once and getting the complete trace, we process step-by-step, updating the trace each time. This allows us to trace through complex algorithms for better understanding and even debugging purposes. The major challenge here was getting the live-stepping functionality to play nicely with the aforementioned drawing mechanism as any changes to the trace had to be idempotent. This was solved by breaking processing up into a series of *ProcessingSteps* where each step was assigned a unique ID. This means that live stepping to the very end of execution will yield the exact same trace as simply calling the process method.

We also provide an additional file – *runner.py* – which offers a command line interface exposing the functionality of the library. For the example given above, we can either run it as a script itself, or we can run it through this interface. This is useful when we want to run and draw graphs in one go and also when we want to live step through the program in an intuitive way – this is enabled by the *LiveStepper* class. Most notably, however, this interface allows us to start a web server (using the standard library module *HTTPServer*) where we can live step through a program and watch the resulting graph update in real time, alongside being able to customise the graph to our liking. This is enabled by the *vis.js* [12] library and some custom JavaScript functions which communicate with the backend server from the browser. The command line interface is implemented with *click* as before.

Finally, to test the library, we re-wrote each of the three stories looked at before and implemented them with this new story-thinking library. This required thinking about the code in a different and unique way and, in some cases, some major elements had to be re-written. This was, however, quite painless due to the simple and user-friendly interface that the library provides to developers. Using these examples, it was possible to test every aspect of the framework, alongside any edge cases and more complex scenarios such as those with threaded agents. Furthermore, an example Jupyter Notebook is provided which showcases some of the functionality of the library.

Below are some images of the previously discussed graphs drawn by the library.

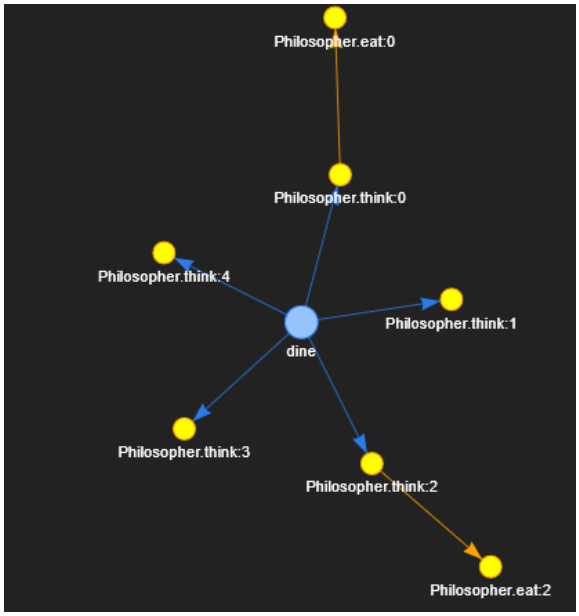

Figure 25: Dining Philosophers graph

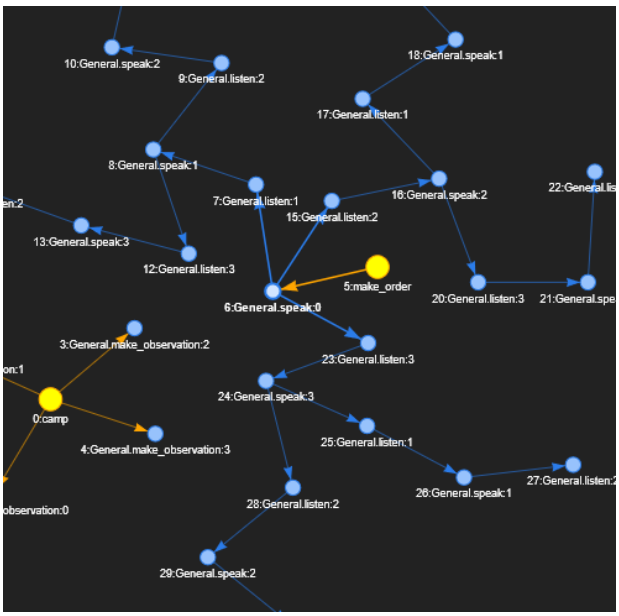

Figure 25: BGP graph

8 References

[1] L. Lamport, R. Shostak, and M. Pease, ‘The Byzantine Generals Problem’, *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, p. 20.

[2] J. Verwolf, *The Byzantine Generals Problem*. 2022. Accessed: Apr. 11, 2022. [Online]. Available: https://github.com/JVerwolf/byzantine_generals

[3] E. W. Dijkstra, ‘Hierarchial Ordering of Sequential Processes’, p. 29, 1971.

[4] ‘Random walk’, *Wikipedia*. Mar. 01, 2022. Accessed: Apr. 12, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Random_walk&oldid=1074697533

[5] *flairNLP/flair*. flair, 2022. Accessed: Mar. 24, 2022. [Online]. Available: https://github.com/flairNLP/flair

[6] M. Honnibal, I. Montani, S. Van Landeghem, and A. Boyd, *spaCy: Industrial-strength Natural Language Processing in Python*. 2020. doi: 10.5281/zenodo.1212303.

[7] S. Talaga, *Narcy: NLP relations extractor and narrative analysis library*. 2021. Accessed: Mar. 24, 2022. [Online]. Available: https://github.com/sztal/narcy

[8] *NetworkX*. NetworkX, 2022. Accessed: Mar. 31, 2022. [Online]. Available: https://github.com/networkx/networkx

[9] *$ click_*. Pallets, 2022. Accessed: Apr. 13, 2022. [Online]. Available: https://github.com/pallets/click

[10] *Yet Another Keyword Extractor (Yake)*. LIAAD - Laboratory of Artificial Intelligence and Decision Support, 2022. Accessed: Apr. 13, 2022. [Online]. Available: https://github.com/LIAAD/yake

[11] *WestHealth/pyvis*. West Health Institute, 2022. Accessed: Mar. 31, 2022. [Online]. Available: https://github.com/WestHealth/pyvis

[12] *vis-network*. vis.js, 2022. Accessed: Apr. 13, 2022. [Online]. Available: https://github.com/visjs/vis-network