# 1. Design and Implementation

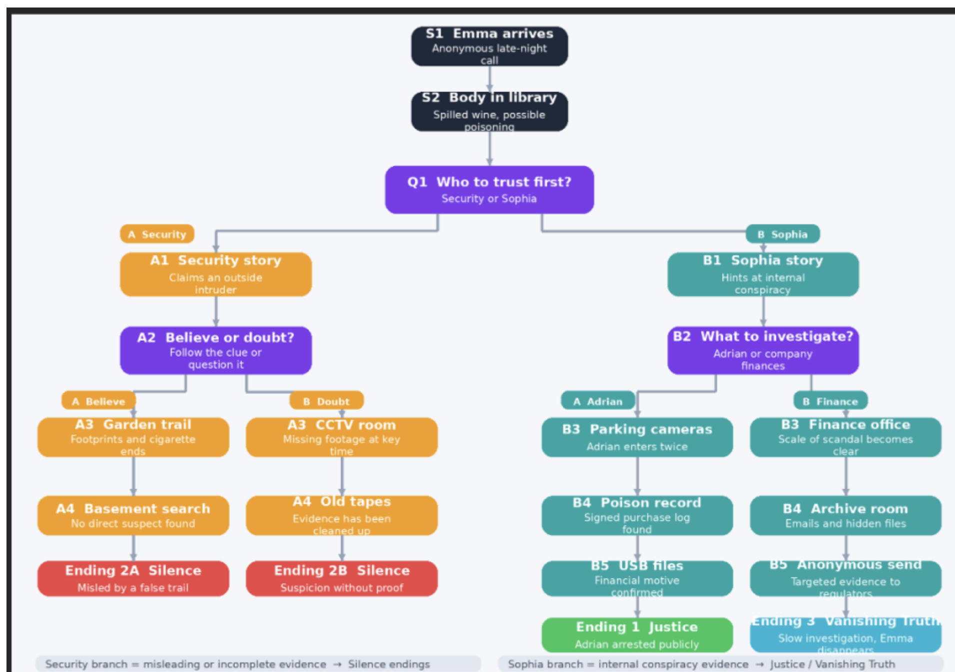
Our team followed an Agile development approach based on the Scrum framework.We had a planning meeting on Wednesday 25th of March where we decided to build a branching narrative mystery game inspired by "Life is strange."

We used a WhatsApp group as our main communication tool.I took the team leader role, which meant keeping track of everyone's contributions and making sure all four sections fitted together properly.

My technical responsibility covered two areas:

- Section 1 - core data structures: StoryNode, Stack, and ClueHeap
- Section 5 - the popup windows: AnalysisWindow and EvidencePanel.

The main design idea was to have multiple endings so the player could get lost in the investigation and use a rewind mechanism to go back and explore different paths.




*(The game structure from the notes)*

Scrum helped me decide the order in which I built each part of the project. I treated StoryNode, Stack, and ClueHeap as separate tasks, and each one was done only when it was tested and fully working with the engine.

This step-by-step approach follows Schwaber and Sutherland (2020) and kept the project always working. After every check, I could spot problems early and fix them before moving on.

## 1.1 Why I built things in this order

I started with the data structures before touching the popup windows, and that was on purpose. The AnalysisWindow and EvidencePanel both need working data to display

So the plan was:

1. Build StoryNode — the container that holds all the information for each scene.
2. Build Stack — for tracking the player's path and making rewind work.
3. Build ClueHeap — for collecting and ranking clues by importance.
4. Build AnalysisWindow — the four-tab dashboard showing algorithm outputs.

I think the correct planning helped us a lot in terms of organising things so we wouldn't run into integration problems later.

## 1.2 StoryNode — the building block of every scene

I use StoryNode because the game needs one clean, main, and consistent place to store everything about a scene. Without it, all the scene data would be allocated across the code, that is harder to manage, and harder to update.

*I used a class instead of a dictionary because a class makes sure every scene has the right fields like node_id, title, and text. With a dictionary, someone could easily misspell a key or forget a field and the code wouldn't catch it.*

```
from typing import List, Dict, Optional, Tuple
```

This library allows me to specify the expected types of variables, parameters, and return values in Python code(Python Software Foundation, 2024)

I chose it to make the code clearer, easier to maintain and more understandable for my teammates.

## 1.3 Stack — making rewind work

The idea is simple:

- the player goes S1 → S2 → Q1_SECURITY.
- Each time they move forward, we push the old scene onto the stack.
- If they hit rewind, we pop the most recent one off and go back there.

I could have just used a normal Python list for this — append() and pop() already do the same thing.

*But the point was to implement it as a proper data structure like we learnt in IFP, not just use list methods.*

**I use copy_list() and to_list()** functions to return a copy of the whole stack as a list, which I added so the Analysis and Evidence panels could read the path history without accidentally modifying the real stack

The engine uses **is_empty()** before every rewind attempt — if the stack is empty, the rewind button gets enabled. Simple, but it prevents crashes.

```
class Stack:
    def __init__(self) -> None:
        self._data: List[str] = []

    def push(self, item: str) -> None:
        self._data.append(item)

    def pop(self) -> Optional[str]:
        if self.is_empty():
            return None
        return self._data.pop()

    def peek(self) -> Optional[str]:
        if self.is_empty():
            return None
        return self._data[-1]

    #week 16 stuff


    def is_empty(self) -> bool:  3 usages
        return len(self._data) == 0

    def size(self) -> int:
        return len(self._data)

    def copy_list(self) -> List[str]:  1 usage
        return list(self._data)          #added

    # alias kept for compatibility
    def to_list(self) -> List[str]:  1 usage
        return self.copy_list()
```

## 1.4 ClueHeap — sorting clues by importance

Throughout the game the player discovers clues at different scenes, and each clue has an importance rating from 1 to 10.

The basic idea of a heap is that it keeps things in a rough order — not fully sorted, but the most important item is always at the top. When you add a new item, it opens itself into the right place in O(log n) time, which is faster than inserting into a sorted list at O(n) (Cormen et al., 2022).

**I considered two approaches:**

- A sorted list using bisect.insort, which gives O(n) insertion due to shifting but O(1) access to the top element;
- Python's heapq, which gives O(log n) insertion and is simple to use.

*I chose the heap because it's both simpler and more efficient in this case.*

I use **get_sorted_clues()** because I want to sort and read clues safely without changing the original heap that stores them.

It prevents data loss and keeps the heap untouched while clues are still in order.

```
def get_sorted_clues(self) -> List[Dict]:  1 usage
    copy_heap = list(self.heap)
    heapq.heapify(copy_heap)
    result = []
    while copy_heap:
        neg_imp, time_f, _, cid, name, desc = heapq.heappop(copy_heap)
        result.append({
            "id": cid, "name": name, "description": desc,
            "importance": -neg_imp, "time_found": time_f,
        })
    return result
```

## 1.5 AnalysisWindow — the algorithm dashboard

The hardest part was styling the tabs to match our dark theme. Because **this was not covered in IFP** — I learnt it independently.

I had to understand how ttk.Style works:

- First, calling **style.configure()** to set base properties like background colour and font on the "TNotebook.Tab" widget
- Then I used **style.map()** to change those properties depending on the tab's state — so when a tab is selected, the background and text colour change to show it's active. Without this, all the tabs looked the same and you couldn't tell which one was open.

```
nb = ttk.Notebook(self.win)
nb.pack(fill="both", expand=True, padx=16, pady=8)

style = ttk.Style()
style.theme_use("default")
style.configure( style: "TNotebook",     background=_BG_DARK,  borderwidth=0)
style.configure( style: "TNotebook.Tab", background=_BG_PANEL, foreground=_FG_DIM,
                padding=[12, 6], font=_FONT_SMALL)
style.map( style: "TNotebook.Tab",
          background=[("selected", "#2a2a2a")],
          foreground=[("selected", _ACCENT)])
```

The whole AnalysisWindow is basically a dashboard that shows the outputs of the four algorithms.

ZhengJiang implemented in Section 3 (DFS, BFS, insertion sort, and topological sort).

The idea was to make the algorithms visible — instead of them just running silently in the background, the player (and the marker) can see what they produce.

## 1.6 EvidencePanel — the clue viewer

My other popup class is EvidencePanel. This one inherits directly from Toplevel using super().__init__(parent), rather than wrapping a Toplevel object like AnalysisWindow does.

I chose inheritance because EvidencePanel is a window — it needs to call **self.title(), self.configure()** , which is cleaner than having a self.win() attribute and calling everything through that.

```
class EvidencePanel(Toplevel):  1 usage
```

I use **refresh()** because the Evidence Panel needs a single, reliable way to update itself every time something in the game changes.

Here's the idea in simple terms:

- It makes sure the panel always shows the correct clues for the currently selected mode.
- It prevents bugs like leftover text, outdated clues, or mismatched sorting.

```
def refresh(self) -> None:  2 usages
    if self._sort_mode.get() == "priority":
        clues = self.engine.get_prioritised_clues()
        header = "Ordered by importance (most critical first):\n\n"
    else:
        clues = self.engine.get_timeline_clues()
        header = "Ordered by discovery time (insertion sort):\n\n"

    self._text.configure(state="normal")
    self._text.delete( index1: "1.0", END)
```

# 2. Testing and Maintenance

The program was developed and tested in PyCharm.

- To run from a clean install:
- install Python 3.12.0
- run pip install Pillow

All code uses only the Python standard library plus Pillow. No other external libraries are required.

## 2.1 What I checked and the ways of checking

**1. Stack / rewind test:** Show the actual node IDs on the stack and the result of popping:

- Path played: S1 → S2 → Q1_SOPHIA. Stack contents: ["S1", "S2"]. Pressed rewind → returned to S2. Stack contents: ["S1"]. Pressed rewind again → returned to S1. Stack empty, rewind button turned off.

**2.Evidence panel sort mode test:** Show the radiobutton switch changes the order:

- After playing to B5_USB, five clues collected. Priority mode displays: "USB Financial Files" (10), "Victor's Body" (10), "Adrian's Car Log" (8)... Timeline mode displays same clues in collection order: "Victor's Body", "Guard's Testimony", "Sophia's Testimony"... Switching back to Priority refreshed correctly with no leftover text.

**3. ClueHeap** — equal importance tie-breaking:

- Added C9 (importance 10, found at time step 5) and C0 (importance 10, found at time step 1). **get_sorted_clues()** returned C0 first, then C9. Confirmed that when importance is equal, the clue found earlier appears first (time_found tie-break).

---

## 2.2 Issues I faced and how I solved them

1)Python’s heapq is a min-heap — it puts the smallest number first. But I want the highest importance first.

- The fix is simple: I store the importance as a negative number. So importance 10 becomes -10, which heapq treats as the smallest (i.e. highest priority).

```
entry = (-importance, time_found, self.counter, clue_id, name, description)
```

2)I always had troubles with sizes and colours of the texts.I decided to structurise this part.

What I did:

- Before writing the actual classes I defined shared colour and font constants at the top — things like _BG_DARK = "#111111" and _ACCENT = "#e6a817".

```
# Shared colour / font constants for popup windows
_BG_DARK   = "#111111"
_BG_PANEL  = "#1a1a1a"
_FG_MAIN   = "#ffffff"
_FG_DIM    = "#aaaaaa"
_ACCENT    = "#e6a817"
_ACCENT2   = "#58a6ff"
_FONT_MONO  = ("Courier", 14)
_FONT_SMALL = ("Arial", 15)
_FONT_BODY  = ("Arial", 15)
```

3)I also added lines 957-960 so that the "analysis" interface can be opened multiple times. Because I found that previously, once it was opened, it couldn't be opened again

```
956
957        def _on_close(self) -> None:
958            global _analysis_window
959            _analysis_window = None
960            self.win.destroy()
```

```
TEST 1: Stack
---------------------------------------------
Created empty stack. Size: 0
Pushed: S1, S2, Q1_SOPHIA
Stack contents: ['S1', 'S2', 'Q1_SOPHIA']
Size: 3
Popping in LIFO order:
  1st pop: Q1_SOPHIA
  2nd pop: S2
  3rd pop: S1
Pop on empty stack: None
PASS: Stack works correctly — last in, first out.
The rewind system can safely go back through visited scenes.
======================================================================
TEST 2: ClueHeap (priority ordering + equal importance)
---------------------------------------------
Added 3 clues:
  Guard's Testimony    — importance 3
  Sophia's Testimony   — importance 7
  Victor's Body        — importance 10
get_sorted_clues() result (should be highest importance first):
  [C0] Victor's Body — importance 10/10
  [C6] Sophia's Testimony — importance 7/10
  [C1] Guard's Testimony — importance 3/10
PASS: Clues sorted correctly by importance (10 → 7 → 3).
---------------------------------------------
What happens when two clues have the SAME importance?
---------------------------------------------
Added 2 clues both with importance 8:
  Adrian's Car Log     — found at step 3
  Archive Documents    — found at step 6

get_sorted_clues() result:
  [C7] Adrian's Car Log — importance 8, found at step 3
  [C11] Archive Documents — importance 8, found at step 6

PASS: When importance is equal, the clue found earlier comes first.
```

(The way I tested Section 1)

**In terms of maintenance :**

- Shared constants at the top of Section 5 mean I can change the whole colour scheme by editing a few lines instead of hunting through hundreds of lines of code.
- If someone wanted to add a new algorithm in the future, they'd just need to create a new frame with **_make_frame(),** populate it, and call **_add.text()** — the rest of the tabbed structure handles itself.
- The EvidencePanel's **refresh()** method already handles any number of clues, so it would scale naturally if the story got bigger.

To further improve **robustness**, future versions could:

- Include automated tests for DFS, BFS, and topological sort functions to verify correct outputs after updates;
- Effectively handle missing image files with a fallback solid-colour background instead of silently skipping the load

## 3. Reflection and Next Steps

Overall, the project went well. I was happy with Wu and Sylvie's contributions — Wu finished his part early which gave me more time to think about how to integrate his work with mine. Mehli found his task challenging, and because I waited for his results rather than checking in early, I had to rush integrating his section which meant less time for testing my own code. Next time I'd handle that differently, maybe by using a more structured sprint system with clear deadlines per week.

Compared to classical Scrum, where roles are clearly allocated and meetings are arranged on a fixed schedule, we only had 5 in-person meetings set up randomly. Our experience shows that even though you can get the work done on time this way, you end up spending more time solving problems than you would with a more structured process.

One thing that went well was the separation between the GameEngine and the UI. Because all the game logic lives in ZhengJiang's engine, my popup windows only needed to call methods like **engine.run_dfs_analysis()** and format the output. This made testing much simpler, however, on the other hand it limited how I could organise my part of the code, since I had to rely strictly on the methods he provided and repeatedly go through his implementation to fully understand how the engine worked.

Working on the data structures helped me understand why choosing the right structure matters, not just how to implement them. For example, using a heap for clue prioritisation means every new clue slots into the right place in O(log n) time, rather than re-sorting the entire list at O(n log n). Implementing the Stack for the rewind system taught me how a simple LIFO (last-in-first-out) structure can replace what I thought would need complex logic. Similarly, designing StoryNode with optional fields like clue and is_ending showed me the value of building flexible data containers rather than creating separate classes for every scene type.

Working with tkinter was an entirely new experience for me - learning how the program responds to user actions, such as clicking a button to call a specific function and understanding how to build graphical interfaces is a skill I believe will prove valuable in my future development work.

For future versions, I would:

- Add the ability to export analysis results to a text file so the player could save their investigation.
- Build a graph visualisation using matplotlib or networkx to show the story tree as an interactive diagram, which would be a big addition but would make the Analysis window much more visual.
- Implement a save/load system using JSON serialisation so the player can save their progress (visited nodes, collected clues) and resume the investigation later.

## References

Python Documentation (2024a) heapq — Heap queue algorithm. Available at: https://docs.python.org/3/library/heapq.html (Accessed: 17 May 2026).

Python Documentation (2024b) tkinter.ttk — Tk themed widgets. Available at: https://docs.python.org/3/library/tkinter.ttk.html (Accessed: 17 May 2026).

Python Software Foundation (n.d.) tkinter — Python interface to Tcl/Tk. Available at: https://docs.python.org/3/library/tkinter.html (Accessed: 16 May 2026).

Schwaber, K. and Sutherland, J. (2020) The Scrum Guide. Available at: https://scrumguides.org/ (Accessed: 18 May 2026).

Van Rossum, G., Warsaw, B. and Coghlan, A. (2001) PEP 8 — Style Guide for Python Code. Available at: https://peps.python.org/pep-0008/ (Accessed: 17 May 2026).

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2022) *Introduction to Algorithms*. 4th edn. Cambridge, MA: MIT Press.

Python Software Foundation (2024) *typing — Support for type hints*. Available at: https://docs.python.org/3/library/typing.html (Accessed: 17 May 2026).