University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Animation of tombstone diagrams

Michal Broos

April 20, 2021

Abstract

Tombstone diagrams are used to illustrate how programming language processors work and how their composition can manipulate software to reach its execution. However, the traditional tombstone diagrams suffer from several problems which is why an improved notation, called J-diagrams, has recently been introduced. Through our background research we found that there was no software for animating tombstone diagrams that could be used in educational settings. Hence, the aim of this project was to design and implement a graphical user interface for animating J-diagrams. Using an iterative development process, we first identified the individual requirements for the software. We then transformed these requirements into a design after considering different alternatives. We implemented the software using PySide2 and finally, evaluated it through usability heuristics and in a user study. We found that all participants thought animation was a useful educational tool. The functionality we implemented was rated on average at 4.88 out of 5 and the design at 4.38 out of 5. Two key findings were that the majority of participants found J-diagrams straightforward and that they would use our software if it were incorporated into the Programming Languages course at the University of Glasgow.

Acknowledgements

We would like to thank:

Dr Ornela Dardha for proposing this project and the engaging supervision sessions;

Ms Uma Zalakain for providing mind-stretching suggestions during the supervision;

the Qt and PySide2 developers for creating a framework beyond our imagination;

the composers of the Xenoblade soundtracks thanks to whom our mind kept sane even during the darkest Shadow of the Lowlands.

Contents

1 Introduction

1.1 Motivation

Language processors, more concretely translators and interpreters, are an important part of computing science. Without language processors, it would not be possible to execute programs unless they were written directly in machine code. Every computer scientist should have at least a basic understanding of translators and interpreters. Hence, it is important to have a clear means of explaining how they work and how their composition can manipulate software to reach its execution. A common way of illustrating these concepts is to use a diagrammatic system called tombstone diagrams. Despite being well-established, tombstone diagrams, or T-diagrams for short, are not perfect. J-diagrams have recently been proposed as a redesign which solves the problems of T-diagrams, thereby providing improved readability. Creating an interactive interface that would animate J-diagrams could prove useful for educational purposes.

1.2 Aims

The aim of the project is to design and implement a graphical user interface which will animate J-diagrams. The user will be able to create an arbitrary diagram composed of translators, interpreters and machines or choose a diagram from the list of predefined examples. The examples will demonstrate a varied range of concepts involving programming language processors such as cross-compilation or bootstrapping. For both user-created and predefined diagrams, the animation will demonstrate how the individual components of the diagram are manipulated. The software will be implemented using PySide2 (Qt). The usefulness of J-diagrams and the implemented software will be evaluated through a user study. The successful implementation will culminate in the software being used as a part of the Programming Languages course at the University of Glasgow. It will also be released to the computing science public.

1.3 Structure

This dissertation is structured as follows.

Chapter 1 – Introduction

- Motivation for the project
- Aims of the project
- Dissertation structure

Chapter 2 – Background

- Introduction of the relevant concepts
- Demonstrating how J-diagrams solved the issues of the traditional T-diagrams
- Exploration of animation as an educational tool
- Related software research

Chapter 3 – Requirements

- Project requirements categorised using the MoSCoW prioritisation technique
- Identification of the target audience

Chapter 4 – Design

- Designing the individual software elements
- Combining the individual elements into the overall design

Chapter 5 – Implementation

- Implementation overview
- Implementation of the individual software components

Chapter 6 – Evaluation

- Demonstrating how usability heuristics were transformed into the software
- Presentation of the user study and its results

Chapter 7 – Conclusion

- Summary of the project
- Personal reflection on the project
- Suggestions for future work

2 Background

In this chapter, we first introduce the language processors and how they relate to our project. We then present the history of tombstone diagrams which culminated in the proposal of J-diagrams. We explain why J-diagrams were proposed, how they have improved the traditional tombstone diagrams and why they have the potential to succeed. We then explore animation as an educational tool and identify what is a good animation. Finally, we present the findings of our related software research.

2.1 Language processors

Before we turn our focus onto the main point of this work, tombstone diagrams and their animation, we shall briefly introduce the topic of language processors. Language processors are a crucial part of computing science because without them, it would be impossible to execute software unless it was written in the particular machine code. There are two main types of language processors - **translators** and **interpreters**.

Watt and Brown (2000) define a **translator** as a program that accepts a source code written in one language, the **source language**, and produces a semantically-equivalent code expressed in another language, the **target language**.
The same authors define an **interpreter** as a program that accepts a source code written in a certain language, the **source language**, and runs that code immediately.
The apparent difference in these definitions is the fact that interpreters do not include any target language. This is because interpreters *do not* translate the source code, which is the key distinction between the two types of language processors.

Both translators and interpreters are programs themselves and thus have to be expressed in some language. This language is referred to as the **implementation language**. Together, the **source**, **target** and **implementation languages** are three key terms which we will use throughout this dissertation with their meanings as just explained.

Translators can be further classified according to the type of source language and target language involved in the translation. Table 2.1 shows this classification. Interpreters can also be divided into categories. However, no such distinction is necessary for this project.

Table 2.1: *The classification of translators.*

Translator type	Source language	→	Target language
Compiler	high-level	→	low-level
Assembler	assembly	→	machine code
Decompiler	low-level	→	high-level
Disassembler	machine code	→	assembly
Transpiler	high-level	→	high-level

Although Watt and Brown (2000) make the difference between translators and interpreters obvious, not all literature follows this explicit approach. For example, Sebesta (2007) describes interpreters insufficiently because they do not provide clear explanations, while Lagerstrom (2003) is plainly wrong because they state that interpreters translate the source code.

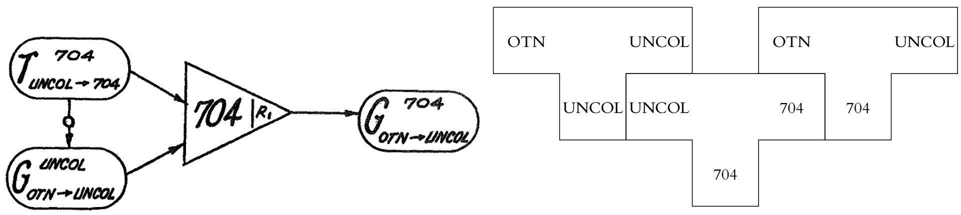
So far, we have introduced 2 main types of language processors. There is one more type, **machines**, which, as Watt and Brown (2000) point out, can be understood as "interpreters implemented in hardware". Similarly, interpreters can be thought of as "machines implemented by software". While not ignoring this relationship, we view a machine, whether abstract or real, as an entity used for running a program expressed in a particular **source language**.

To conclude, there are three takeaways from this section which shall be turned into project requirements.

- Translators, interpreters and machines should be supported.
- There should be a clear distinction between translators and interpreters.
- The classification of translators as shown in Table 2.1 should be applied.

2.2 From T-diagrams to J-diagrams

Tombstone diagrams are used to illustrate language processors and how their composition can manipulate a program to reach its execution. The origins of the concept date back to the diagrams used during the UNCOL project (Strong et al. 1958). T-diagrams themselves were introduced by Bratman (1961) and extended by Earley and Sturgis (1970). Figure 2.1 illustrates these two notations with a practical example.



(a) The predecessor of the first T-diagram *(b) The first T-diagram*

Figure 2.1: *The compiler translating from UNCOL to 704 and running on the 704 machine is used to compile the OTN to UNCOL compiler expressed in UNCOL, thereby producing the latter compiler capable of running on a 704 machine. (a) and (b) illustrate the same process.*

T-diagrams can be created from four unique shapes corresponding to translators, interpreters, machines and programs as shown in Figure 2.2. The composition of pieces can be horizontal, vertical or diagonal with the key restriction being that two languages which meet together must match. Figure 2.3 demonstrates horizontal and vertical composition. We shall talk about diagonal composition shortly.

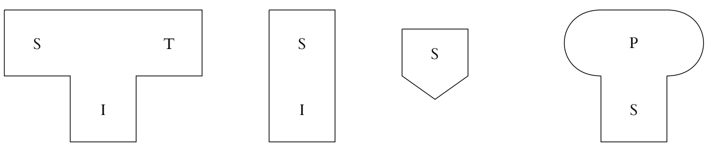
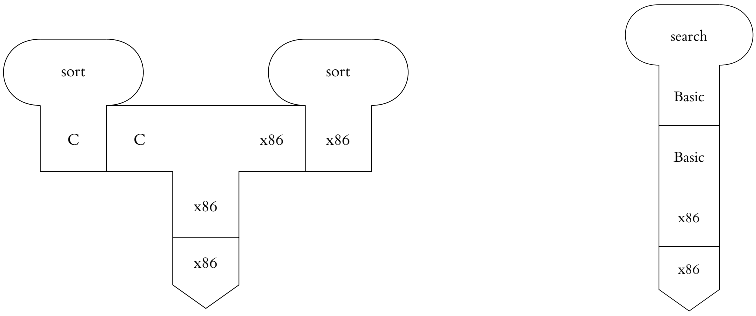


Figure 2.2: *Four shapes used in T-diagrams representing, from left to right, a translator from S to T implemented in I, an S interpreter implemented in I, a machine capable of executing S and a program P written in S where S = source language, T = target language and I = implementation language as described in Section 2.1.*



(a) Compilation *(b) Interpretation*

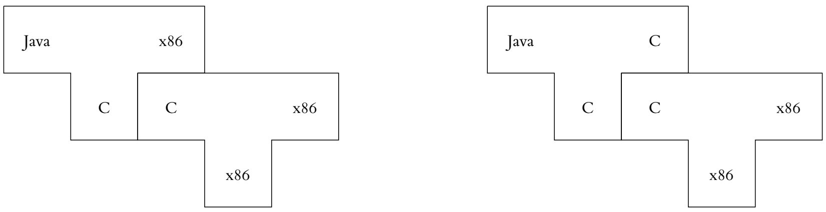
Figure 2.3: Examples of horizontal and vertical compositions in compilation and interpretation. (a) shows a sort program written in C being compiled to x86 by a compiler running on an x86 machine. (b) shows a search program written in Basic being run by an interpreter running on an x86 machine. Notice that in both cases the neighbouring languages match which is a requirement for valid composition of pieces.

Despite being nearly 60 years old, T-diagrams are still used today, especially as a teaching aid. Other authors (Burkhardt 1965; Rosin 1977; Slansky and Finkelstein 1968) came with their own notations but neither was able to replace T-diagrams. The most recent redesign of T-diagrams, titled J-diagrams, has been proposed by Wickerson and Brunet (2020) after identifying the following problems with T-diagrams.

Problem 1 Diagonal composition of two translators is unclear. This is because three languages meet at two interfaces but it is only two of these languages that are required to match. The two matching languages must be the implementation language of the left piece and the source language of the right piece. Figure 2.4 shows how this problem can occur. Note that this problem is also apparent in Figure 2.1b where all three languages that meet are UNCOL.

Problem 2 There is no distinction between the operands and the result of the composition. For example, in Figure 2.1b, it is not clear that the translator on the right is the result of composing the left translator with the middle translator. All three translators appear to be equivalent.

Problem 3 The T shape representing translators is symmetric which means it is possible to compose two translators in a way that appears valid, but in fact it is invalid. The leftmost diagram in Figure 2.5 illustrates this. Note that it seems this composition is used in Figure 2.1b. However, it is not the composition being present but rather a manifestation of **Problem 2**.



(a) Clear matching of languages *(b) Unclear matching of languages*

Figure 2.4: Diagonal composition of two translators. In each case, the composition is valid but ***Problem 1*** *is apparent in (b).*

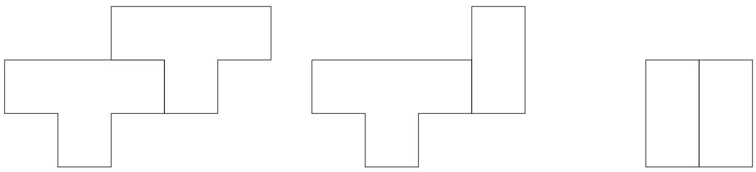


Figure 2.5: Examples of invalid composition in T-diagrams illustrating ***Problem 3****.*

There are further problems caused by the choice of shapes for both translator and interpreter pieces which we identified in addition to those mentioned by Wickerson and Brunet (2020). They follow the same logic as described in **Problem 3** and thus we shall consider them as belonging to that problem. The middle and the rightmost diagram in Figure 2.5 are examples of such further problems. Note that in the middle diagram, if we moved the interpreter to the left of the translator, the composition would be valid because it would be a case of translating an interpreter. Once again, it is the symmetry causing the problem.

Wickerson and Brunet (2020) solve **Problem 1** and **Problem 3** by designing the individual pieces such that no ambiguous composition is possible. **Problem 2** is solved simply by not showing the composition results in the diagrams. J-diagrams can be created from three unique pieces corresponding to translators, interpreters and machines as shown in Figure 2.6. Compared to T-diagrams, there is no piece for programs. The pieces allow only two types of composition - horizontal and diagonal. Figure 2.7 demonstrates composition in J-diagrams using three practical examples. Note how two pieces only ever connect at one interface, thereby solving **Problem 1**. Also note how invalid compositions are impossible to create in J-diagrams (Figure 2.8), thus solving **Problem 3**. Other than solving the problems, J-diagrams are also more concise and easier to understand than T-diagrams. This can be observed in Figure 2.10.

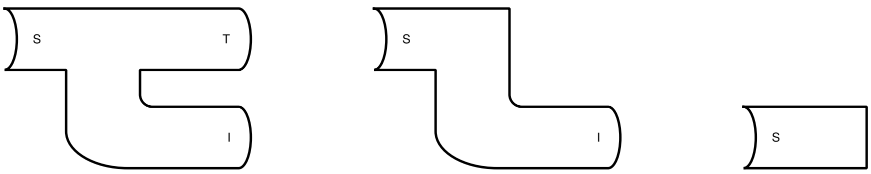
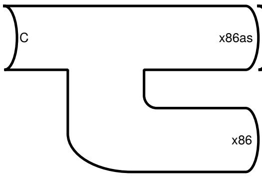


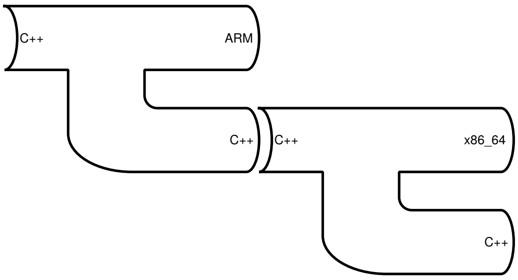
Figure 2.6: Three pieces used in J-diagrams. The pieces are presented in the same order and have the same meaning as the pieces of T-diagrams in Figure 2.2. Note that there is no program piece in J-diagrams.

In a sense, one could think that J-diagrams are just another notation which will be forgotten. However, the main advantage of J-diagrams over the other notations is that J-diagrams are not a completely new system. They are based on the established system of T-diagrams. Hence, transitioning from one to another is seamless. Additionally, Wickerson and Brunet (2020) not only identified the problems of T-diagrams but also demonstrated how the problems are solved with J-diagrams. Rosin (1977) also identified problems with T-diagrams but never explicitly stated how they are resolved. Moreover, problem identification was not thorough. Although the issue of a possible but an illegal composition (**Problem 3**) is fixed by Rosin's notation, the key problem which persists is that of matching languages (**Problem 1**). It is not clear which languages of composed elements should match. For T-diagrams, this problem only occurs when composing two translators in which case three languages meet at two interfaces. In Rosin's notation, this problem is intensified because as many as four languages meet at one interface and the problem occurs when a translator and an interpreter or two interpreters are composed (two translators cannot be composed directly in one line in this notation). Figure 2.9 illustrates this.

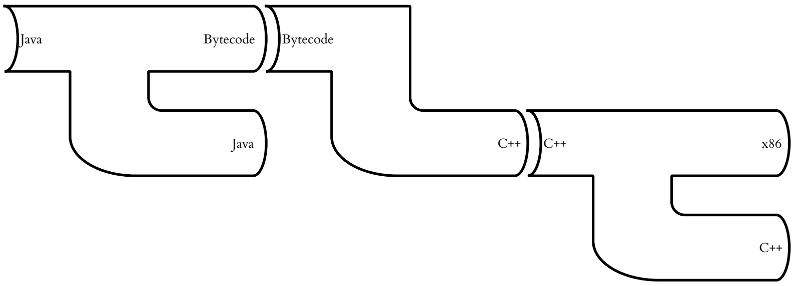
Considering the advantages of J-diagrams, we chose them as the notation for our project.



(a) Horizontal composition of two translators



(b) Diagonal composition of two translators



(c) Horizontal and diagonal composition including two translators and an interpreter

Figure 2.7: *Examples of horizontal and diagonal composition in J-diagrams. (a) shows a two-stage compilation of a C program to x86. (b) shows compiling a C++ compiler from C++ to x86_64. (c) is an example of Java compilation. It is clear from these examples how J-diagrams solve* ***Problem 1*** *of T-diagrams.*

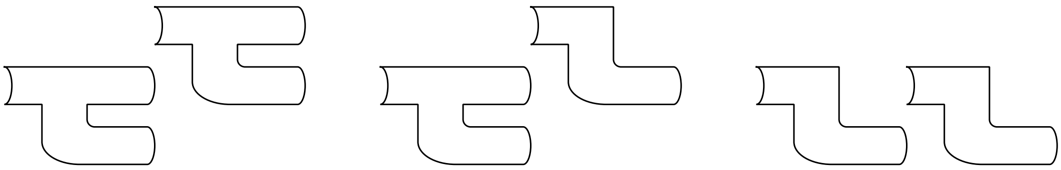
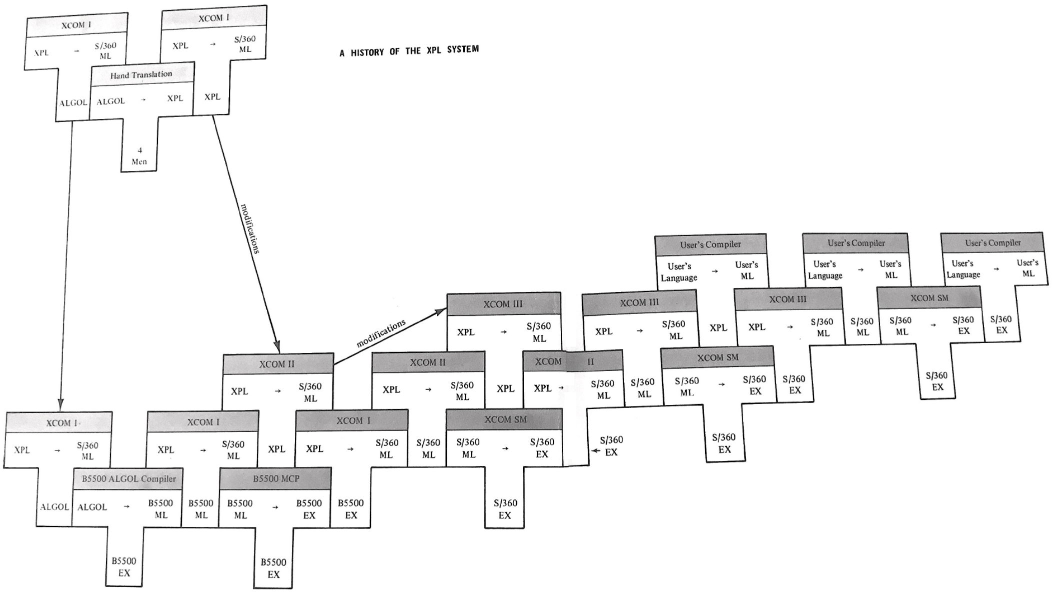
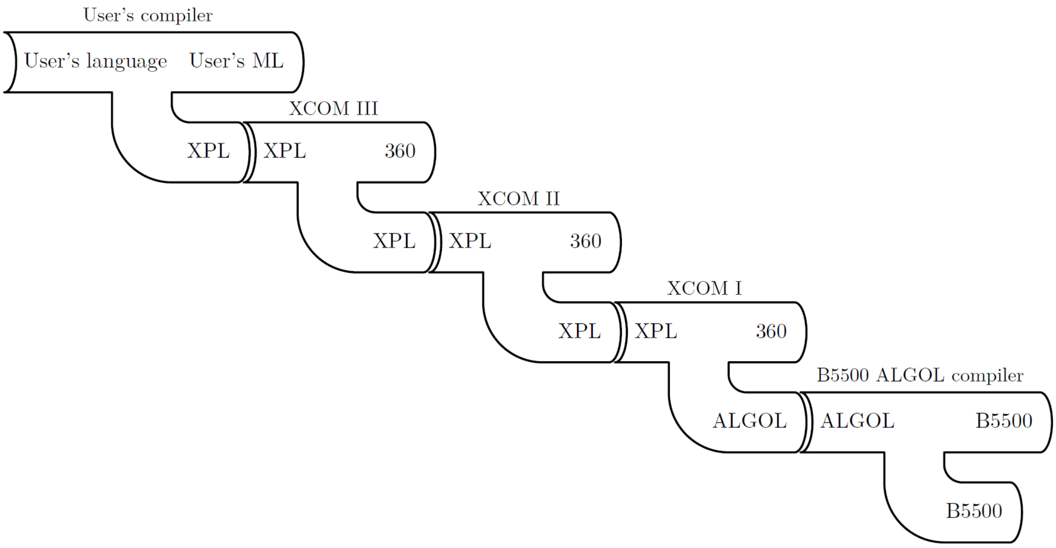


Figure 2.8: *Equivalents of the T-diagrams from Figure 2.5 showing how J-diagrams solve* ***Problem 3****.*

Figure 2.9: *A diagram from Rosin (1977). Notice how four languages meet at the second and the third piece from the left (a translator and an interpreter), as well as at the two middle pieces (two interpreters). This is an intensification of* ***Problem 1****.*



(a) The original representation of the XPL's history



(b) A more concise representation of the XPL's history

Figure 2.10: The history of XPL expressed as a (a) T-diagram and (b) J-diagram. Source: Wickerson and Brunet (2020).

2.3 Animation as an educational tool

Using animation as an educational tool is a vast topic that one could dedicate an entire project to. The topic as a whole is beyond the scope of this dissertation. However, before proceeding, we must answer one key question regarding animation. We need to know if there is any point in animating tombstone diagrams.

According to Ross and Grinder (2002), in the sphere of computer science education, algorithm animation also referred to as visualisation is most common. The same authors state that "visualizations play a key role in providing insights into important concepts". The authors of the two well-known animation tools for data structures and algorithms (Galles 2011; Halim 2011) express a similar message of helping students to understand data structures and algorithms better. Despite this, there have been questions about effectiveness of algorithm animation. As a reaction to this, Hundhausen et al. (2002) conducted a meta-study of 24 experimental studies about algorithm visualisation effectiveness. The authors found that algorithm visualisation is educationally effective, not as a mechanism for transferring knowledge to students but rather as a "vehicle for actively engaging students in the process of learning". Although the research focused on algorithm visualisation, we believe the findings are applicable to animation of tombstone diagrams as well. Not only are both subsets of educational software visualisation, but animation of algorithms also involves some kind of diagram illustrating the underlying data structure. For example, when animating the longest common substring problem, the suffix tree shown in Figure 2.11 is a type of diagram. The positive outcome of this research means that continuing with our project is meaningful.

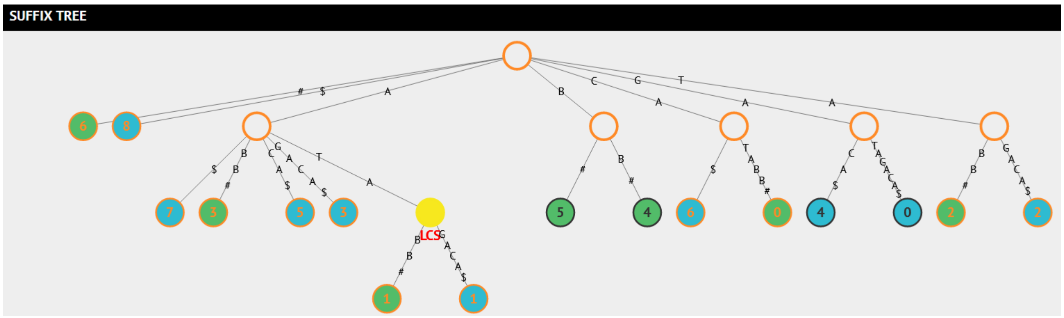


Figure 2.11: *Suffix tree diagram used in the animation of the longest common substring algorithm on VisuAlgo.*

Mernik and Zumer (2003) confirmed these findings in their research closer to our topic. The authors developed a visualisation tool for teaching compiler construction and observed the following educational benefits of animation:

- Keeping students active in their learning
- Helping students to develop mental models
- Stimulation of exploratory learning by providing immediate feedback
- Different learning style and speed are supported
- Increased motivation to learn
- Better understanding of concepts

Although our work does not involve constructing compilers, we would expect similar benefits from a tool for animating tombstone diagrams. This is because the tool would represent just another level of abstraction above, that is, not showing how compilers are constructed but how they are composed with other language processors.

Fleischer and Kučera (2002) summarised suggestions of several authors regarding what concerns a good algorithm animation. We picked the following key points applicable to our project which shall be transformed into project requirements.

- The representation of diagrams should be uniform throughout the tool.
- User-created diagrams should be supported.
- Animations should be simple without any elaborate graphics. Colour should be used for highlighting the current step.
- Textual explanations should be included.
- Player controls bar should be available.
- It is sometimes helpful to show different stages at the same time. Hence, it should be possible to animate multiple stages independently.

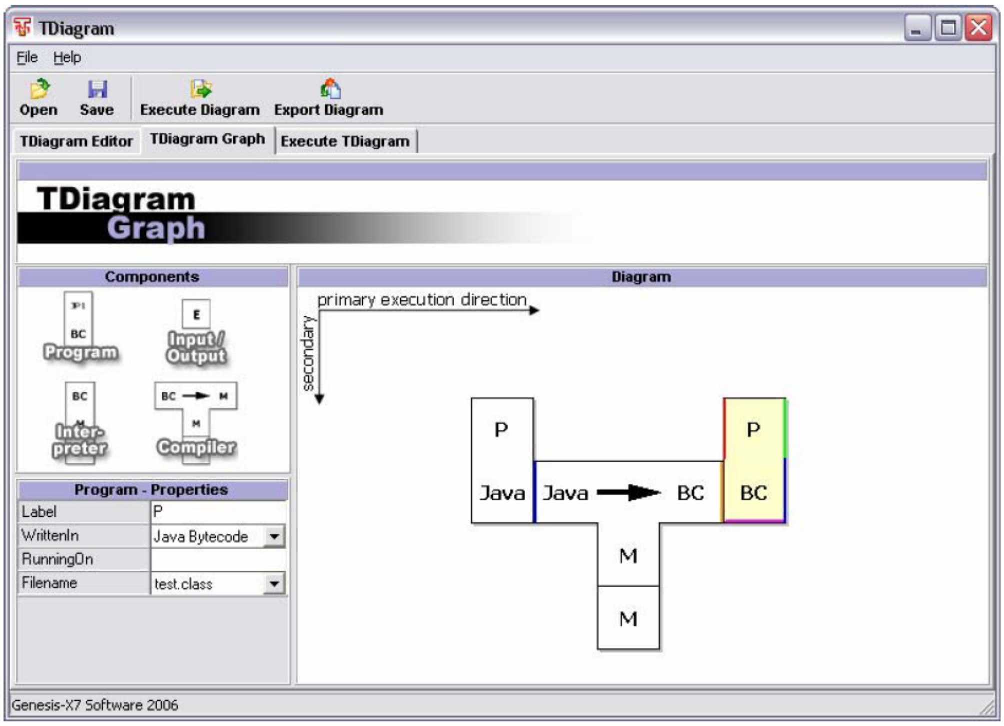
2.4 Related software

Finally, before proceeding with requirements gathering, we had looked for any software using tombstone diagrams. The point of this part of background work was to find an answer to the question “Is there a piece of software for animating tombstone diagrams? If not, what related software is there?”

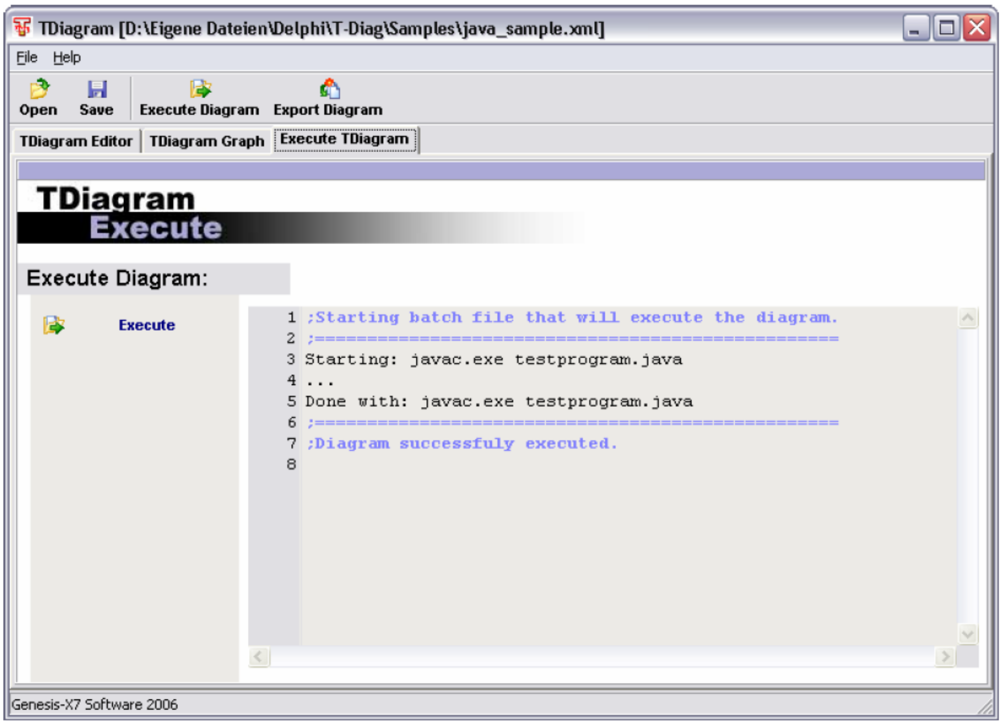
Searching for “tombstone diagram” and “t-diagram” on GitHub, GitLab and Google revealed several projects, most of which were unrelated to our type of diagrams. We noticed a degree of clashing terminology with other fields, e.g. p-T diagrams in Chemistry or the tombstone phenomenon in circuitry, and even within Computing Science, e.g. tombstones as markers for dead code analysis. There are a few LaTeX macros and a package for drawing tombstone diagrams which we slightly modified and used for drawing T-diagrams in this chapter (Jakobsen 2017). There is also an extension for the diagramming software Dia which adds the four shapes used in T-diagrams (ter Horst 2014). Both of these offer creation of T-diagrams but not their animation. Additionally, the Dia extension is cumbersome to use because the shapes include only one language text field by default. It is sufficient for machines but for every interpreter and translator, which by definition include two and three languages respectively, text fields have to be added manually. Having found Dia, we looked at one of the most popular online diagramming tools called diagrams.net (draw.io previously). This tool does not support T-diagrams and the only way to create them is by combining the letter T and I outlines with text boxes. Such a solution is of course very inefficient.

The most related piece of software we found is called TDiag. It was developed by Hielscher (2006*a*) as one of the six parts of the learning environment AtoCC (**A**utomata to **C**ompiler **C**onstruction). Since AtoCC is a multipurpose system for teaching Language Theory and Automata, T-diagrams are not its main focus. It is possible to create T-diagrams in TDiag and the program supports connecting pieces together as well as diagram validation. However, there is no animation. The main point of TDiag is to allow users to execute diagrams attached to local files. Figure 2.12 shows the interface of TDiag when creating a diagram and the corresponding diagram execution screen. Since TDiag uses the traditional T-diagrams, it suffers from the problems described in Section 2.2. We identified one additional serious problem. The author uses the same shape to represent the programs and interpreters. This proprietary notation is apparent in the Components category in Figure 2.12(a) and in the diagram in Figure 2.13.

Having conducted related software research, we concluded there was no software for animating tombstone diagrams, thereby validating the point of our project.



(a) Creating a diagram in TDiag



(b) Execution of the diagram from (a)

Figure 2.12: *Two screens of TDiag, the program for creating and executing T-diagrams, which is a part of the AtoCC learning environment. Source: Hielscher (2006b).*

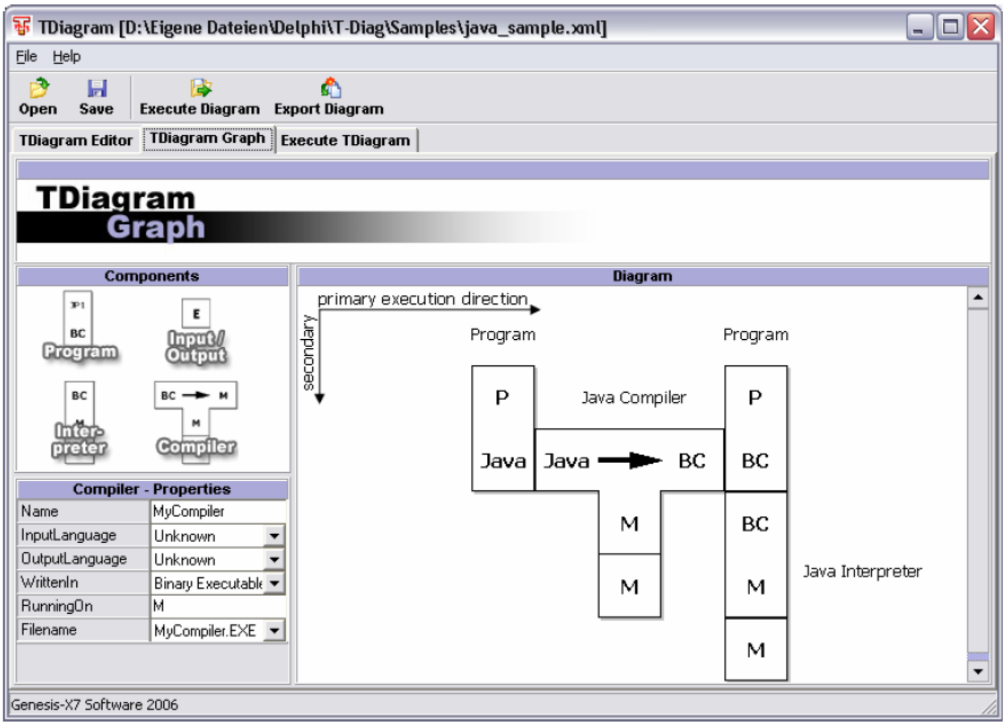


Figure 2.13: Diagram in which it is apparent the programs and interpreters are represented by the same shape in TDiag.

3 Requirements

The requirements for the project were gathered by two means. The key features requirements were collected from conversations with Dr Ornela Dardha, a researcher in the Formal Analysis, Theory and Algorithms section at the University of Glasgow and a lecturer of the Programming Languages course. Other requirements either arose from our background research or were thought of and proposed ourselves based on our prior experience with usability heuristics.

3.1 MoSCoW

Whenever a new requirement occurred, it was categorised using the MoSCoW prioritisation technique (Agile Business Consortium 2021). The output of this process was a live list of requirements which kept being updated throughout the development until the final version was produced.

Must have

1. A software which can be launched quickly without a complicated installation
2. Create arbitrary diagrams composed of translators, interpreters and machines
3. Use the J-diagrams notation
4. An intuitive drag and drop interface
5. Load diagrams from the list of predefined examples
6. Animate both user-created and predefined diagrams
7. A list of draggable diagram pieces grouped in categories (using the classification from Table 2.1 for Translators)
8. An option to add custom diagram pieces, i.e. pieces with user-defined languages, not new shapes
9. Connect two pieces together as in a jigsaw puzzle when neighbouring languages match
10. Disallow connecting pieces when neighbouring languages mismatch and inform the user about the problem
11. Overlapping pieces must not be allowed
12. Allow repositioning of pieces in the diagram scene while ensuring connections stay valid
13. Utility operations on individual pieces - delete, increase and decrease height
14. Animation scene which, once started, is unaffected by changes in the diagram scene
15. Explanatory textual output accompanying the animation
16. During the animation, the currently being animated piece must be highlighted
17. The ability to play, pause and replay the animation
18. Clear distinction between translators and interpreters during the animation
19. Use colour blind friendly palette throughout the software

Should have

20. A multi-platform software
21. A software guide - video, text or both
22. The ability to go through the animation step by step forwards and backwards
23. The ability to restart the animation and to bring it to the end instantly
24. The possibility to open multiple animation scenes, all independent of each other
25. Keyboard shortcuts for all key features
26. Predefined examples extensible by the user
27. The ability to distinguish diagram pieces by naming them
28. Import and export of diagrams
29. Exported diagrams should use a common format which is human-readable and editable
30. A way of clearing the entire diagram scene
31. A diagram scene grid to help visualise positioning
32. The ability to turn the diagram scene grid on and off
33. Automatic save and load of user-defined languages/pieces providing persistence across executions
34. The ability to remove all user-defined languages/pieces, thus reverting back to the default pieces only

Could have

35. The ability to rename diagram pieces
36. Change the speed of animation
37. Export diagram scene as an image and support both vector and raster graphics file formats
38. Highlighting the position where a piece will appear once dropped
39. Change the colour scheme, e.g. a dark theme
40. Image preview when choosing from the list of predefined diagrams
41. A dedicated home screen before proceeding to the diagram scene

Won't have this time

42. Diagram scene zoom in and out
43. Collapse and expand all categories in the list of diagram pieces
44. Allow selection of multiple pieces at once so that actions could be performed in bulk
45. Highlighting the mismatched language(s) rather than the whole piece when an invalid connection is made
46. Undo and redo functions linked to the diagram scene operations
47. Copy and paste functions for diagram pieces
48. Dedicated, rather than programmatically produced, text output for predefined diagrams
49. Structure the format of exported diagrams such that it is implementation-independent so that the diagrams could be used in other software
50. Video guide embedded directly rather than showing a YouTube link

3.2 User stories

The requirements listed in Section 3.1 could all be expressed in the form of user stories. Rather than repeating ourselves, we summarise these requirements into four high level user stories describing the types of users which our software targets.

User 1

As a student who is taking/has taken the Programming Languages course,
I want to be able to interact with and animate tombstone diagrams,
so that I can improve my understanding of the concepts which are difficult to understand purely from the written theory.

User 2

As a a lecturer of the Programming Languages course,
I want to have a tool for visualising language processors,
so that I can make the course content more engaging and easier to understand.

User 3

As a researcher writing a paper/book about language processing,
I want to create diagrams of the systems I am presenting,
so that I can include them in my writing, thereby helping readers to understand those systems.

User 4

As a student who has never taken the Programming Languages course,
I want to explore a tool illustrating compilation and interpretation which is easy-to-use and offers guidance,
so that I can improve my computing science knowledge.

4 Design

In this chapter, we describe the individual elements of our interface design. We provide design alternatives whenever applicable and give reasons for rejecting them. To illustrate the designs, we present wireframes created in the wireframing tool Balsamiq. The final section describes how the individual elements were combined into an overall design.

4.1 List of pieces

The point of this interface element is to allow the user to choose pieces which they want to compose together to create a diagram. Each piece is represented as a single row divided into three columns. The entries in the columns correspond to the source, target and implementation languages for the given piece. Since interpreters have no target language and machines have no target and implementation languages, the entries in those cases are "-", expressing the meaning "not applicable".

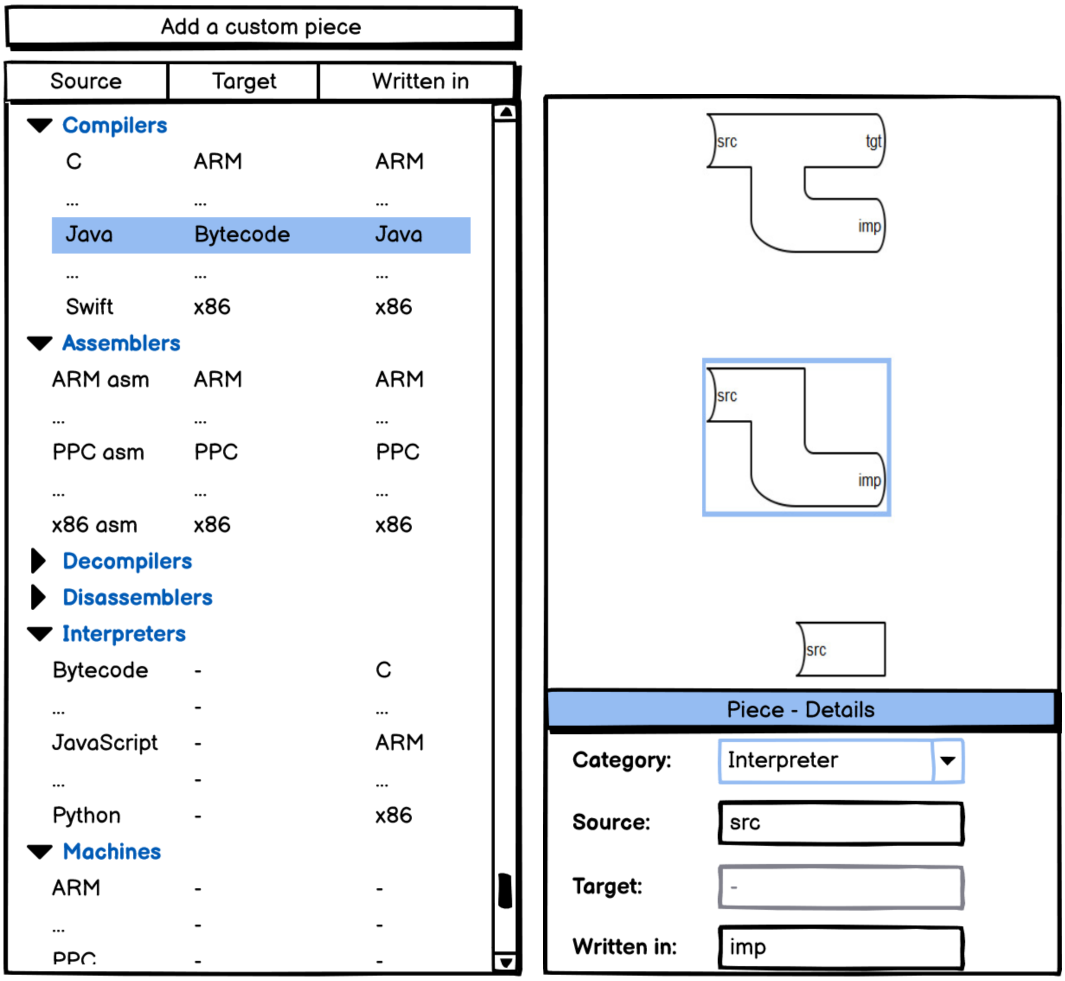
We considered representing languages using their file extension instead of full names. For example, ".py" instead of "Python". This would make the list less wordy because file extensions are generally shorter than languages names. However, we rejected this alternative based on the fact that it would be less explicit and unclear.

Another idea regarding the presentation of languages was to merge into one row the translators with the same source and target languages but a different implementation language, i.e. *(sourceA == sourceB) AND (targetA == targetB) AND (implementationA != implementationB)*. Merging of the interpreters would follow the same logic based on the matching source languages. The advantage of this approach would be having a shorter list to scroll. On the other hand, the user would have to be prompted for the implementation language every time, regardless of the prompt design (e.g. a dialog with buttons, drop-down menu or text input). We rejected this idea because we believed the convenience of not having to specify the implementation language each time outweighed the convenience of having a shorter list. Additionally, disconnecting the implementation language from the other languages could cause unnecessary confusion about the meaning of the pieces.

The pieces are sorted in the ascending alphabetic order first by the source language, then by the target language and finally by the implementation language. By pressing a keyboard letter while having an arbitrary piece selected, it is possible to scroll instantly to the first piece whose language starts with the given letter.

The pieces are grouped in categories which can be collapsed and expanded. There is one category each for interpreters and machines. The categories of translators follow the classification from Table 2.1. An alternative design would be to have only a single category for translators or to have no categories at all. However, finding the desired pieces in such a design would be unnecessarily difficult. We considered different ways of distinguishing the category headings from the pieces. For example, we tried changing the text colour, highlighting the headings and making them bold. We found the combination of the first and the last option to be the least intrusive but still noticeable.

The design allows the user to select a piece and drag it onto the diagram scene (Section 4.3). Conversely, dragging pieces within the list is not possible to avoid confusing the user. Our design also does not show the shapes of the pieces in the list. We considered this unnecessary because the user should already be familiar with them and even if they were not, it would only take dragging a piece from each category onto the diagram scene to become familiar with the shapes. We initially experimented with a design which did show the shapes. The list of pieces in this graphical approach contained nothing but the three types of shapes used in J-diagrams. The shapes could also be dragged onto the diagram scene and the languages would be specified either by typing or choosing from a group of drop-down menus below the shapes similar to TDiag. We did not take this idea further because we considered selecting the languages in this way more time consuming than finding them directly in the list of pieces. Figure 4.1 shows two wireframes, one for our final design and the other for the abandoned graphical approach design. The final design contributed to the realisation of Requirement 7.



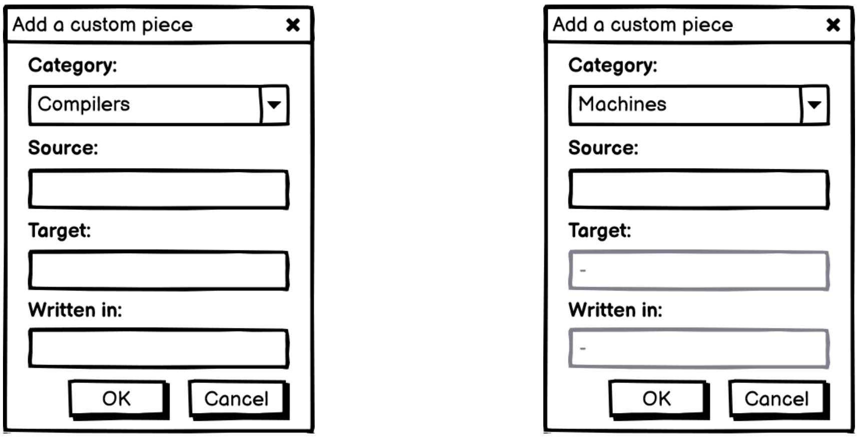
(a) Final design *(b) Abandoned design*

Figure 4.1: *Wireframes showing two design alternatives for the list of draggable pieces. Note that we used "Written in" instead of "Implementation" to be more concise and explicit.*

4.2 Adding custom pieces

There is one more element in Figure 4.1a which we did not mention in the previous section, the **Add a custom piece** button. This button is linked to a dialog window which allows the user to add their own pieces, meaning pieces with user-defined languages, not shapes. Without this, the user would be restricted to the list of predefined pieces. Since it is impossible to include all languages by default, this is a must-have feature. An alternative design would be a simple "+" button. We rejected this alternative because it would be less explicit. Additionally, we were not designing for mobile devices and thus there was enough space to work with.

Our design in this case is straightforward. The dialog asks the user to choose a category for the piece and type in its languages. Obviously, the available categories match those in the list of pieces. Choosing the category, especially for translators, is a good practice for novice computer scientists because it encourages them to think what category is correct for the given language processor. An important design decision in this step was disabling the input boxes for the "not applicable" languages of interpreters and machines. Another design decision we had to consider was where the added pieces should appear in the list. One option was to add them below the predefined pieces (while still respecting the categories, of course) and another option was to insert them based on the alphabetic ordering. This is most likely a matter of personal preference but we chose the former to be consistent. The last design choice was to include automatic scrolling in the list to the newly added piece and highlighting it. Figure 4.2 shows the final design of the dialog in this step which contributed to the fulfilment of Requirement 8.



***(a)** Adding a new compiler*   ***(b)** Adding a new machine*

***Figure 4.2:** Wireframes showing the final design of the dialog for adding user-defined pieces. Note the disabled input boxes for the "not applicable" languages in (b).*

4.3 Diagram scene

This element allows the user to create arbitrary diagrams composed of individual pieces. It can be thought of as a canvas on which the corresponding shapes and languages are painted. The painting happens as soon as a piece is dropped onto the diagram scene.

Since diagrams are created by connecting pieces together, we had to come up with an appropriate design solution. A common technique to achieve this in diagrammatic software is to apply so-called "grid snapping". This is indeed how we designed our diagram scene. We split the entire scene into a grid of smaller rectangles and the pieces are automatically positioned using these rectangles. To improve visualisation when positioning the pieces, we included the grid in the scene. The design of the pieces follows the J-diagrams notation with the dimensions chosen such that both translators and interpreters take up four grid rectangles and the machines take up a single rectangle. We chose the snapping based on the top left corner of a piece. This means that when a piece is dropped for the first time, i.e. it is dragged from the list of pieces, its top left corner is aligned within the grid rectangle where the mouse pointer was at the time of dropping. The precise alignment point is an implementation detail. This behaviour is illustrated in Figure 4.3.

Another advantage of grid snapping is that it makes repositioning of pieces easy. The only difference in design between repositioning and dropping a piece for the first time is that for repositioning, the new position is not determined from the mouse pointer position but from the position of the top left corner of the piece. We made this decision after realising that using the mouse position would be inconsistent since repositioning can be initiated after clicking anywhere on the piece. Of course, an alternative would be to display a repositioning icon in the top left corner of a selected piece and only allow repositioning after clicking on the icon. The mouse position could then be used as it would always be consistent. However, this would reduce usability by reducing the flexibility of repositioning and thus we rejected that idea.

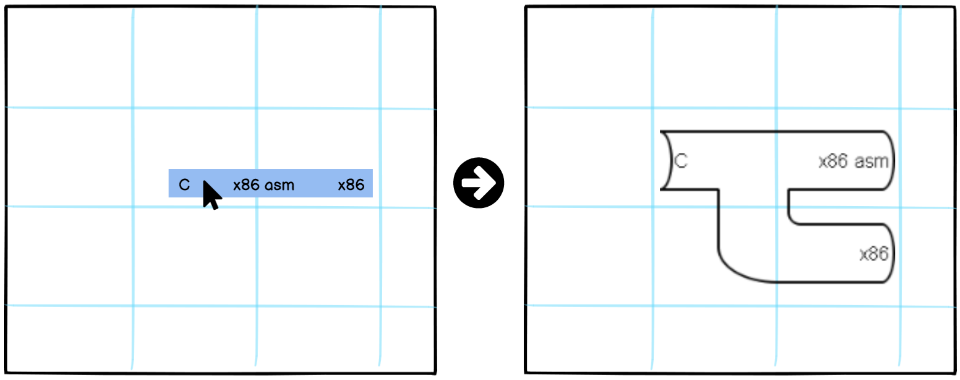


Figure 4.3: *Wireframe illustrating grid snapping when dragging a piece from the list of pieces and dropping it onto the diagram scene. Notice how the top left corner of the piece in the right image is not positioned precisely where the mouse pointer dropped the piece in the left image, but is instead snapped to the grid rectangle containing the mouse pointer.*

Combining the design of grid snapping with the regular design of the pieces means that the pieces connect and align naturally by being placed next to each other at the desired positions. To make the diagrams meaningful, connections are only allowed when the neighbouring languages are identical. We could have designed the scene such that even incorrect diagrams could be created. However, since one of the main purposes of the software is to be educational, such a design would not offer the immediate feedback required to learn. Hence, when an invalid connection is established, the user is informed about the problem through a pop-up dialog and by highlighting the problematic piece. An alternative design would be to highlight just the mismatched languages. We did not take this design further as we did not identify any benefits from it. The user can easily see which languages mismatch even in the first design. If a mismatch happens after dragging the piece from the list of pieces, the piece is deleted. If it is a result of repositioning, the piece is moved back to its previous position. Figure 4.4 illustrates this situation. Overlapping pieces are similarly meaningless. For consistency, such situations are rectified in the same way but with an updated dialog message.

There are a few alternative approaches to the design of positioning and connecting the pieces which we could have taken. Firstly, instead of grid snapping, we could have used free positioning. The user would be allowed to place and reposition pieces freely but they would have to connect

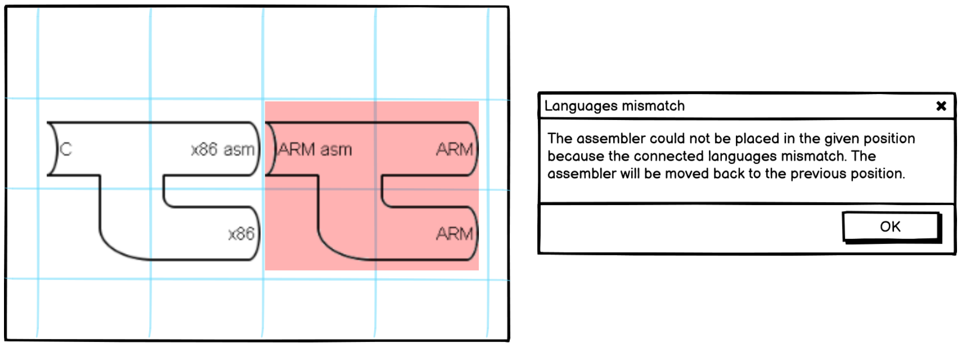


Figure 4.4: *Wireframe illustrating an invalid connection between two translators caused by mismatched languages. Highlighting of the culprit and a pop-up dialog are used to inform the user about the cause of the problem and how it will be rectified.*

them manually, for example, by switching to a connection definition mode and clicking on dots appearing next to the languages only in this mode. In our design, connections arise automatically based on the positions. Secondly, the easiest design to implement would have been one where if a piece were placed, the next one would have to be connected to it. However, such a design would not allow repositioning and it would be impossible to create two or more disconnected diagrams which is useful either for comparing diagrams or when creating diagrams consisting of multiple stages. Finally, an alternative approach still using grid snapping would have been to replace "drag and drop" with "click and click" where two clicks correspond to selecting a piece and clicking elsewhere to place it. All of these alternatives would have been either more restrictive or less intuitive which is why we opted for our design.

In Figure 4.3, we illustrated the process of placing a piece onto the scene. Since we focused on grid snapping at that point, we intentionally omitted one intermediate step between dropping a piece and the piece appearing in the scene. In this step, the user is offered a choice to name the piece. The reason behind this design decision was to give the user the ability to identify individual pieces easily. This is achieved by showing a dialog window with a text input box where the user can type the name. Although we would encourage naming the pieces, we recognised it is not absolutely necessary, especially in simpler diagrams. Hence, we emphasised in the dialog title that the input is optional. Another design decision we had to consider was how to display the names. We chose to show them at the top of each piece where they do not interfere with any connections. Figure 4.5 shows the design of the name input dialog and how the named piece appears in the diagram scene.

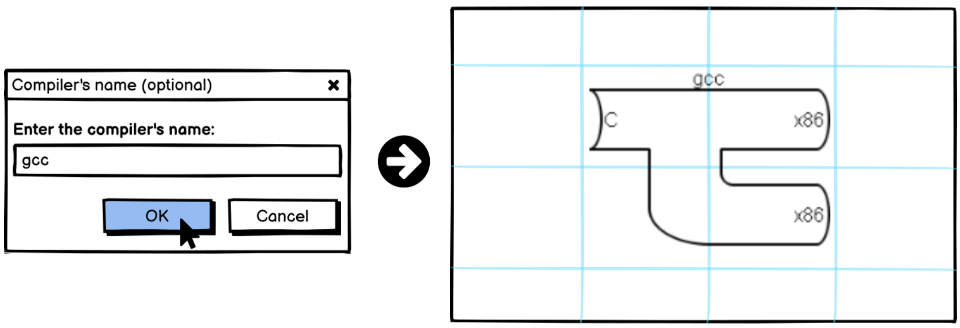


Figure 4.5: *Wireframe showing the name input dialog and how the named piece appears in the diagram scene.*

It was also necessary to design the process of increasing and decreasing the height of the pieces. Changing height was identified as a must-have requirement because when two translators or a translator and an interpreter are composed horizontally, it becomes impossible to connect another piece to the implementation language of the left piece unless its height is increased. Since changing height had to be compatible with grid snapping of pieces, we designed it such that the height of a given piece is increased or decreased by the height of the grid rectangle. In other words, if a default-sized translator or interpreter has its height increased once, it no longer occupies four grid rectangles (2x2) but six (2x3). Note that we did not mention a machine in the previous sentence because changing the height of the machines has no benefit. We ensured that our design accounts for this and informs the user about this through a pop-up dialog.
An alternative design would be to have a "short" and a "tall" default version of translators and interpreters which could be chosen upon dropping a piece. However, this design would still not be sufficient because in more complex diagrams, increasing height more than once is not uncommon and there is no "one-height-fits-all" piece.
Another condition that our design had to meet was that changing height did not result in an invalid connection, i.e. in a languages mismatch. If it does, the user is informed about the problem via a pop-up dialog and the problem is resolved by restoring the previous height.
The action to change height is accessible by right-clicking a piece. Other than increasing and decreasing height, it is also possible to delete the piece from this menu. To give the user flexibility, it is also possible to change height from the menu bar, toolbar or using keyboard shortcuts (more on the design of these in Section 4.4). Figure 4.6 illustrates the design of heightening a piece. Shortening of course follows the same pattern but with an opposite effect on the height.

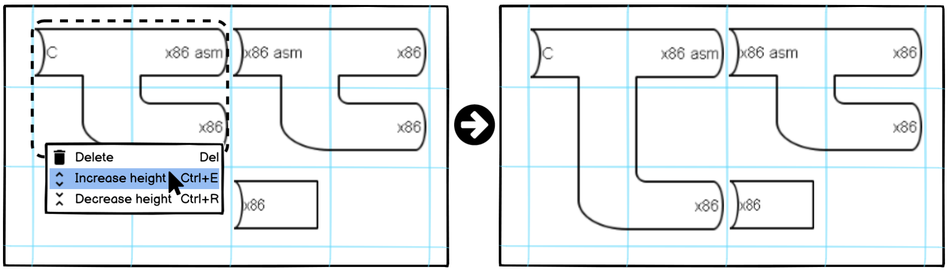


Figure 4.6: Wireframe illustrating the design of changing the height of a piece. Note that on the left, it is impossible to connect the machine to the compiler's implementation language because the machine would overlap with the assembler. Heightening of the compiler as shown on the right solves the problem.

The design described in this section contributed to the accomplishment of Requirements 2-4, 9-13, 27 and 31.

4.4 Menu bar, toolbar and keyboard shortcuts

The logic behind designing the menu bar was to include all available actions other than those relating to drag-and-drop and the list of pieces. Figure 4.7 shows all menu bar categories and their corresponding menus.

In the *File* menu, the user can create a new diagram. If the scene contains a diagram already, a dialog asks the user to confirm they are happy to wipe the old diagram. The user can choose to create a blank diagram or to load a predefined example. The design of this process is illustrated in Figure 4.8. The user can also import and export diagrams. There is nothing specific about the design of these actions other than the need to provide a file name. To keep the design consistent, a dialog must be confirmed to wipe an existing diagram before importing can happen. If the user tries to export an empty diagram scene, they are informed about this in a dialog and are asked to place some pieces first. The exit action is self-explanatory.

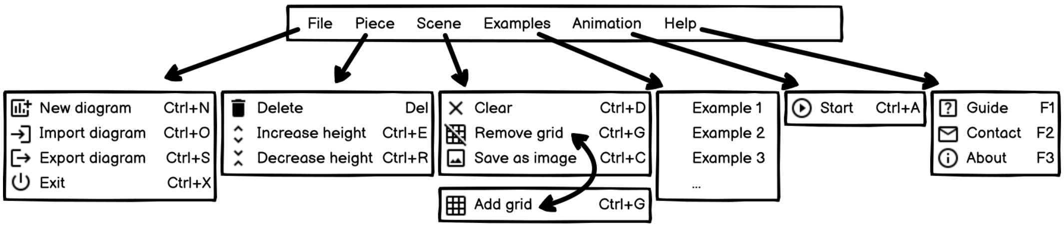


Figure 4.7: Wireframe showing the design of the menu bar and the corresponding menus. Note that in the Scene menu, the grid action changes depending on if the grid is on or off.

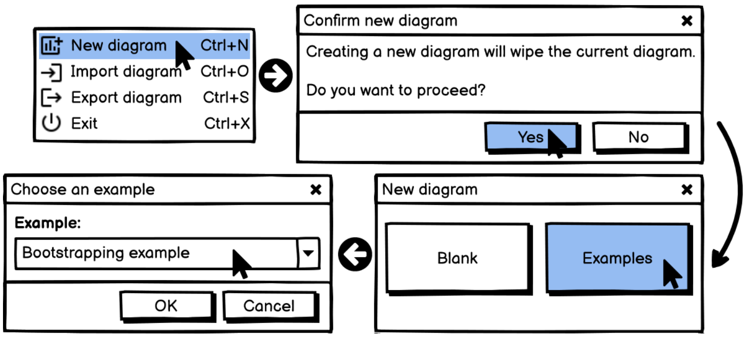


Figure 4.8: Wireframe illustrating the design of creating a new diagram. Upon confirming the last step (bottom left), the chosen predefined example appears in the diagram scene.

The *Piece* menu offers the same actions as when a piece is right-clicked in the diagram scene. We described this design in the last part of Section 4.3.

In the *Scene* menu, the user can clear the diagram scene. This action is designed to delete all pieces from the scene but again, not until the user confirms in a dialog that they are happy to proceed. If the scene is clear already, this action is pointless which is why the user is given feedback about this. When we defined the design of the diagram scene in Section 4.3, we mentioned including a grid. The user could find the grid distracting when not positioning pieces but simply looking at a diagram, trying to understand it. Therefore, we gave the user an option to remove the grid and add it back when desired. We designed the grid action such that when the grid is on, *Remove grid* is displayed and when the grid is off, *Add grid* is shown instead. The corresponding icon also changes accordingly. This can be noticed in Figure 4.7. The user might also want to present their diagrams outside our software, e.g. User 3 from Section 3.2. Because of this, we included in our design an action to save diagrams as images. The file formats which are supported is an implementation detail.

The *Examples* menu is designed to allow quick opening of one of the predefined example diagrams. There are no loading dialogs to go through as the selected example is instantly shown in the diagram scene. A dialog which can appear when opening an example is the usual confirmation prompt to prevent accidentally removing an existing diagram. One thing which the design in Figure 4.7 does not portray is our decision to make the list of examples extensible by the user. The implementation details of this are discussed in Section 5.6.

The only option available in the *Animation* menu is to start animating a diagram. The animation design is described in Section 4.5 and 4.6. The design of the actions in the *Help* menu is simplistic. *Guide*, *Contact* and *About* information are all presented in a pop-up window. We decided to include both a video and a text guide for the software. There is not a more precise way to demonstrate full capabilities of the software than with the video guide. On the other hand, the text guide is useful for quickly looking up specific details. There were two possibilities how to include the video guide - embedding the video directly in the software or uploading it to YouTube and providing a link. We opted for the latter since it meant a smaller size when distributing the software.

We also designed a toolbar offering quick access to all actions from the menu bar except *Exit*, *Examples* and *Help*. The *Exit* and *Help* were excluded because there is no need to access them quickly. We could have included *Examples* using a drop-down button but it would not provide access any more quickly than selecting an example from the menu. For consistency, we used the same icons as in the menus. We included tooltips to make it clear which action each icon represents without having to refer back to the menus. Figure 4.9 shows the design of the toolbar. To enable an even greater efficiency, we designed all actions except loading the individual examples so that they can be triggered with keyboard shortcuts. We chose the shortcuts either by following conventions, e.g. *Ctrl+N* for *New diagram*, or by choosing a letter resembling the action, e.g. *Ctrl+G* for *Remove/Add grid*. All shortcuts can be seen in Figure 4.7.

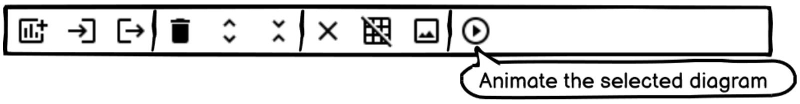


Figure 4.9: *Wireframe showing the design of the toolbar.*

By combining the design decisions presented in this section, we give the user flexibility to choose whichever way of triggering actions they prefer. The design contributed to the realisation of Requirements 5, 21, 25, 28, 30, 32, and 37.

4.5 Animation elements

The design of animating diagrams consists of three elements - **the animation scene**, **the box for displaying textual output** and **the play controls**. The animation scene always displays the entire diagram which is being animated. There is no need for the grid because the pieces are already positioned. The text box is initially empty. The individual animation steps correspond to animating the individual pieces comprising the diagram. We chose to provide visual feedback to the user by highlighting the currently being animated piece. As soon as a piece is highlighted, its description appears at the bottom of the text box. The descriptions of the previously animated pieces remain in the box. An alternative design would be to start with a blank animation scene and make the pieces appear one after another as the animation unfolds. The text box behaviour in this design would remain unchanged. We decided against this design because the individual pieces are not independent. It would not make sense to show a piece and provide its description referring to how it is connected, if those connections were yet to be revealed. Figure 4.10 illustrates our chosen design.

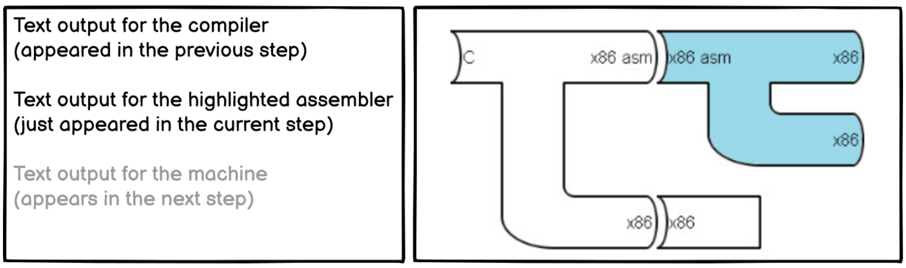


Figure 4.10: *Wireframe illustrating the animation scene and the text output box. The entire diagram is displayed, the currently being animated piece is highlighted and the corresponding text output is displayed.*

Animation can only be useful if the user can control it, hence the need for play controls. We included five play control buttons in our design (Figure 4.11) with the following names and intended functionality.

Restart If an animation has started, whether it is playing, paused or finished, this button brings it back to the start by clearing the highlighted piece and removing all text output. Otherwise, the button does nothing since the animation is already at the start.

Backwards This button makes an animation go back one step by highlighting the previous piece and showing the text output up to and including that piece. It automatically pauses the playing animation. It also allows going back once the animation is finished. If the animation has not started, the button does nothing.

Play/Pause If an animation is not playing, this button appears as Play and it starts or resumes the animation by highlighting the next piece and showing the text output up to and including that piece. There is a short break before moving to the next step; the precise value is an implementation detail. If the animation is finished, the button serves as a replay from the start button. If the animation is playing, the button appears as Pause and it stops the animation at the current piece, keeping the text output and highlighting.

Forwards This button is equivalent to Backwards but in the opposite direction. If pressed after the animation is finished, the animation wraps around to the beginning. If the animation has not started, the button advances it by one step.

Finish This button brings the animation to the end by clearing any highlighting and showing the entire text output. If the animation is finished, the button does nothing.

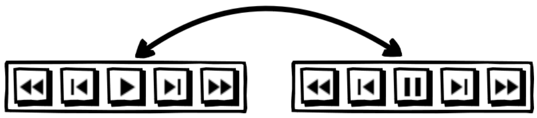


Figure 4.11: *Wireframe showing the animation play controls. Note that the Play/Pause button changes depending on if the animation is playing or not.*

The design presented in this section contributed to the fulfilment of Requirements 15–18, 22 and 23.

4.6 Overall design

Having designed the individual interface elements, we then combined them together into an overall design. Figure 4.12 demonstrates the final design of the window that the user is met with after starting our software. This screen can be thought of as the home screen. We considered an alternative design in which the home screen would be a welcome page similar to that of Eclipse or Visual Studio Code. On this welcome page, the user would be able to select an option to create a new diagram, import a previously exported diagram or load a predefined example. The user would get to the diagram scene and the list of pieces after choosing one of these options. There would also be links to the Help menu actions. We scrapped this design because it offered no benefits. Excluding the welcome page means the user can start creating diagrams or exploring animations straight away rather than being frustrated by having to go through one more step.

For the overall animation design, we made animation accessible in a new window. We considered tabbed interface but that would make it impossible to see the diagram and animation scene at the same time. The final design of the animation window is presented in Figure 4.13. The window is opened after triggering the start animation action introduced in Section 4.4. Animation always requires a precise starting point. This is of course known for the predefined examples. However, for the user-created diagrams, the starting point cannot be inferred because the user can place pieces in any order and position, not to mention multiple diagrams in one scene or multistage diagrams. Therefore, we designed the starting of animation such that the user must first select a piece where the animation should start. The animation then goes through all pieces on the path from the starting point. We considered using the leftmost piece as the starting point but in such cases, it would be impossible to animate disconnected diagrams. Asking for user input was the most flexible solution. If the user tries to start without selecting a piece, they are informed about this via a dialog. Once the user starts the animation, the corresponding diagram is copied from the diagram scene to the animation scene which makes the two scenes independent of each other. Since we identified in our background research that it is sometimes useful to animate multiple stages at the same independently, we extended our design to allow opening of multiple animation windows. Hence, the final design consists of one main window and as many animation windows as the user needs.

The design decisions described in this section contributed to the attainment of Requirements 6, 14 and 24.

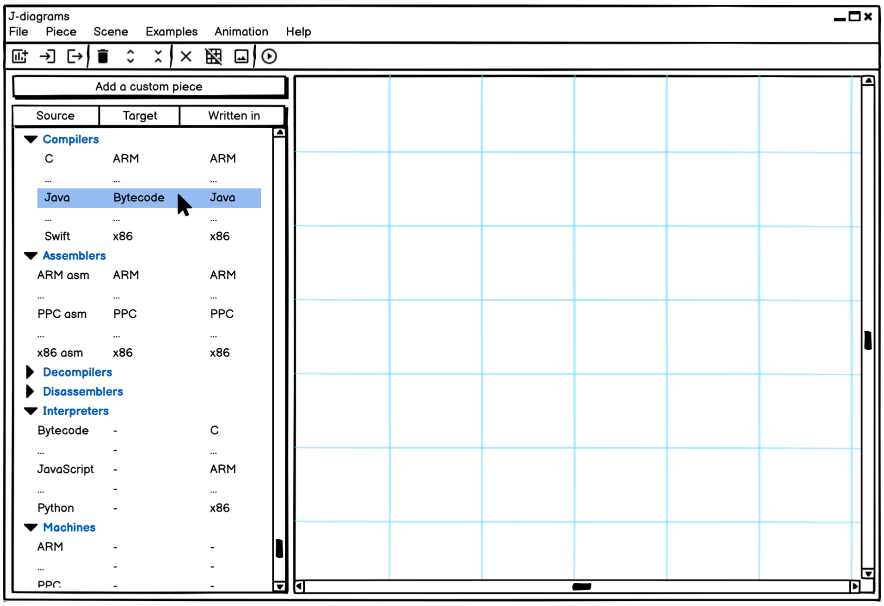


Figure 4.12: Wireframe demonstrating the final design of the main window.

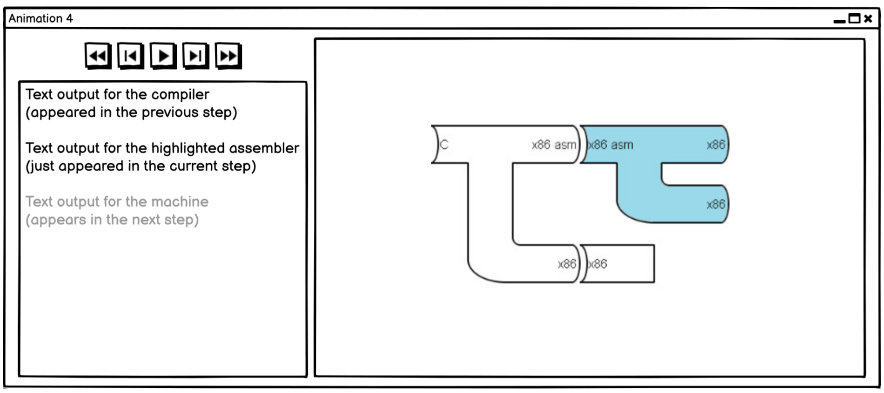


Figure 4.13: Wireframe demonstrating the final design of the animation window. Note that the animation is currently paused, hence the Play button showing.

5 Implementation

In this chapter, we describe how we transformed the wireframes from Chapter 4 into a concrete implementation. We first give a brief overview and then focus on the main components of our software in their own sections.

5.1 Overview

We implemented our software using PySide2 which is a Python binding of Qt, the framework for development of graphical user interfaces. Although we developed our software in Windows, the choice of Qt/PySide2 means that it is possible to run the software on Linux and macOS because the framework is cross-platform. This was a realisation of Requirement 20. It is not unusual for PySide2 applications to be contained in one file. We chose this approach since we were not developing reusable modules. Despite developing in Python, we did not follow the PEP 8 naming conventions. We used camelCase instead to match the PySide2 conventions. Our implementation consists of five classes as described in the next five sections. In our descriptions, we refer to PySide2 classes which always start with Q.

5.2 List of pieces

This feature is a list only in the general English meaning. Implementation-wise, the design maps to a tree with the root being the reference point, the level 1 nodes being the categories and the level 2 nodes, i.e. the leaves, being the individual pieces. We implemented this feature in the *PiecesTree* class which is a subclass of *QTreeView*. Subclassing *QListView* would not allow to include categories. Using *QTreeView* in our implementation meant using the Model-View architecture of Qt. This is similar to the well-known Model-View-Controller design pattern but the difference is that the View and the Controller are combined. We included our model within the *PiecesTree* class as an instance variable because it was not necessary to subclass it. This does not mean the model and view were combined into one or that the design pattern was broken. It simply means we chose to have a single point of reference to the pattern as a whole.

One of the challenges was to decide what pieces, i.e. languages, to include and how to include them. We started with a small sample of manually created pieces whose languages were stored in tuples. The tuples were collected in lists which were the values in a dictionary with the language processor categories as the keys. However, there were so many language processors that it would not be feasible to include them all. Similarly, including only a small seemingly random subset could be questioned by the user. Thus, our logic in the end was to write a list of common architectures (machines) and lists of common compiled, compiled to bytecode, interpreted and transpiled languages. By iterating over these lists in nested loops, we automatically generated the compilers, interpreters and transpilers. We combined the pieces in these three categories with the *x86*, *ARM* and no other machines to avoid too many similar language processors differing only in the implementation language. The decompilers were generated from the compilers by simply reversing the source and the target language. For the assemblers and disassemblers, we adopted a simple approach of translating from and to the assembly language of every included machine. In other words, for every machine, there is one assembler and its mirrored disassembler. For the machines, we did not make a distinction between the 32-bit and 64-bit architectures. It would mean duplicating all pieces and the difference is not crucial for diagramming. In any case, the user has the option to add a piece they need. The dictionary described at the beginning of this paragraph remained as the data structure for storing all default pieces (see Listing A.1). Our automated approach after providing the initial input resulted not only in a variety of the included languages but also in the well-structured categories.

The remaining functions in the *PiecesTree* class deal with adding the pieces to the model such that the view can present them. The steps of this are too Qt specific and full of little details which is why we do not focus on them. The dialog for adding user-defined pieces as shown in Figure 4.2 is also implemented in this class.

5.3 Diagram pieces

As the name suggests, the *DiagramPiece* class is used for representing the individual diagram pieces. The class is instantiated either when a piece is dropped onto the diagram scene, or when a piece is copied onto the animation scene, or when a diagram is loaded from the Examples menu. It is a subclass of *QGraphicsItem*. *QGraphicsItem* provides a basis for creating custom items which can be added to *QGraphicsScene* (our diagram scene, Section 5.4) which is in turn visualised by *QGraphicsView*. These classes are part of the Qt's Graphics View framework.

All three types of a piece, i.e. translators, interpreters and machines, are completely represented by the *DiagramPiece* class. The precise type is specified when instantiating the class as one of the three class variables corresponding to the types. Other than the type, there are straightforward instance variables that identify the piece's category, name, languages, tooltip and geometric properties. There are also Boolean flag instance variables. One to distinguish if the piece is being animated and another one for flagging situations when the piece is in an invalid position caused by overlapping with another piece or having mismatched languages at the specific connection interface. To be able to move the piece back to its previous position after being incorrectly repositioned, we store this position in an instance variable and update it at every repositioning. The most interesting are the instance variables relating to how we keep track of the piece's connections and how we identify what interfaces, if any, two adjacent pieces meet at. For the former, we keep three variables that we refer to as links, one for each type of language (source, target, implementation). Each link holds a reference to another *DiagramPiece* object connected at the specific language/interface. For the latter, there are three variables which store the y-coordinate of each connection interface. If the y-coordinate of source of piece A equals the y-coordinate of target or implementation of piece B, then the two pieces are connected. Since the interpreters and machines have no target and the machines also have no implementation language/interface, it was necessary to set the corresponding variables in those cases to *None*.

The majority of the functions in the *DiagramPiece* class are concerned with defining the geometry of pieces. Painting of pieces is implemented by overriding the paint function of the superclass. The unique shapes of the three types of pieces are defined as paths (*QPainterPath*) and returned from their separate functions. The paths are implemented as sequences of connected lines and arcs in the clockwise direction with the initial start in the top left corner. Each change of the angle in the path corresponds to a new line or arc. Although unique, the paths had to be defined such that the connection interfaces between different shapes are aligned. Other than painting the shape, the *paint* function also deals with painting the name and languages of the piece. It also includes conditionally executed painting of the selection rectangle when a piece is selected and highlighting when a piece is in an invalid position or being animated. We also overrode the *shape* function of the superclass which is used for detection of overlapping pieces. This function uses another overridden function which defines the bounding rectangle of the entire piece.

We only identified the requirement to change the height of translators and interpreters after implementing their standard shapes. Thus, we had to update the original implementation. Our solution was to add the *heightMultiplier* instance variable and update the y-coordinates of all lines and arcs that are moved by heightening or shortening. Since we designed changing height in line with grid snapping, the updated y-coordinates are obtained as the product of the height of the grid rectangle, i.e. half of the default height of the piece, and the *heightMultiplier*. This simply means that all lines and arcs that are required to move are shifted by one grid rectangle down for heightening and up for shortening. A crucial function we had to include in relation to this is a predefined setter which updates the y-coordinate of the piece's implementation connection interface based on the current value of the *heightMultiplier*. The setter is called every time a heightening or shortening call occurs. Omitting this would result in incorrectly identifying the piece's connections.

5.4 Diagram scene

The diagram scene is implemented in the *DiagramScene* class which is a subclass of *QGraphicsScene*. Just as *QGraphicsScene* is responsible for the management of *QGraphicsItem* objects, our *DiagramScene* manages the *DiagramPiece* objects. The class contains only a few instance variables which are mainly defined for ease of access and having a single reference point to the stored values. Conversely, it includes 24 functions which corresponds with its management task.

The first group of related functions are reimplementations of the drag and drop event handlers of the parent class. These functions are responsible for handling only the initial drag and drop of pieces from the list, not any subsequent repositioning which we discuss in the next paragraph. The key action when a drag is entered (*dragEnterEvent*) is to accept only the right type of data, i.e. a piece in our case. Once the drag is being moved within the scene (*dragMoveEvent*), we ignore the event if the mouse pointer goes over an already placed piece. This is one part of preventing overlapping pieces. There was no need to reimplement leaving the scene during an active drag (*dragLeaveEvent*) because it is ignored by default. The handler called when a piece is dropped (*dropEvent*) has the most involved implementation out of these functions. It includes the following steps.

1. Get the current position and piece data from the event.
2. Ask the user to name the piece by displaying a dialog (Figure 4.5).
3. If the user confirmed the dialog:
    4. Instantiate *DiagramPiece* based on the piece data.
    5. Check if the piece was not dropped in the bottom row or the rightmost column of the scene. This would mean having a half of the piece out of bounds. Change the coordinates to in bounds, if necessary.
    6. Place the piece and select it.
    7. Check if the piece does not partially overlap with another piece. If it does, call the invalid position handler which deletes the piece.
    8. Call the *connecting pieces check* algorithm.

In the last step, we referred to the *connecting pieces check* algorithm. This is an algorithm we designed as a solution to the problem of verifying that the pieces are connected correctly and establishing and updating their links. We discuss the algorithm shortly.

The second group of related functions are reimplementations of the mouse event handlers which are triggered when the mouse input originates in the diagram scene. Unlike the drag and drop event handlers described in the previous paragraph, these functions are responsible for handling the repositioning of pieces. When the mouse is clicked (*mousePressEvent*), we only accept the event if it was a left-click as right-clicking a piece is handled in *DiagramPiece*. Since multiple selection of pieces was not a high priority requirement, we also ignore clicking with *Ctrl* pressed. When the event is accepted, the key step is to get the piece at the click position and if it exists, store this position in the piece's previous position instance variable. This allows the piece to be moved back if its repositioning is invalid. When the mouse is moving over the scene (*mouseMoveEvent*), the event is simply propagated. We could not employ the same approach of ignoring the event when moving over a piece as in *dragMoveEvent* because it would be impossible to reposition a piece by moving it over other pieces. Finalising the repositioning by releasing the mouse button (*mouseReleaseEvent*) includes the following steps.

1. If there is a selected piece, i.e. the piece being repositioned:
    2. Reposition the piece. If the piece is repositioned out of bounds, move it in bounds. In this case, we had to cover all four scene edges because there are more ways to achieve out of bounds repositioning.
    3. Check if the piece does not overlap with another piece. If it does, call the invalid position handler which moves the piece back to its previous position.
    4. Call the *connecting pieces check* algorithm but only if the repositioning changed the position, i.e. *current position != previous position*. Calling the algorithm would be wasteful otherwise.

For both *mouseReleaseEvent* and *dropEvent*, the last two steps call the same functions. We use the parameter *initialDrop* in both functions to distinguish between the two situations. Having introduced the two cases when the *connecting pieces check* algorithm is called, we can now present the algorithm as pseudocode in Listing 5.1. The algorithm is fairly complex because of having to deal with all the different ways how pieces can be connected and how the corresponding situations have to be handled.

```
def checkConnectingPieces(self, piece, initialDrop):
  """
  Algorithm which verifies if the placed piece forms valid connections and
  handles both valid and invalid cases. In either case, the algorithm ensures
  the correct links are stored inside the pieces. The parameter piece is the
  placed piece that triggers the algorithm call. It is the only diagram piece
  we refer to as 'piece' in the pseudocode. All other pieces are referred to as
  'neighbour' and there can be zero or more neighbours. For brevity, in a few
  steps we use p and n where p == piece, n == neighbour. We refer to three
  connection interfaces of piece and neighbour as Source, Target and Implem.
  There are three Tasks, each of which corresponds to checking the connection
  at the specific interface. The zeroth Task is an edge case. Arrows -> and <-
  in comments show the direction of valid connections with piece being the
  starting point. The initialDrop parameter is passed to other calls within the
  algorithm to distinguish between the initial drag and drop and repositioning.
  """

  # Get y-coordinate of every connection interface of piece
  yPieceSource = y-coordinate of Source of piece
  yPieceTarget = y-coordinate of Target of piece if it exists, else None
  yPieceImplem = y-coordinate of Implem of piece if it exists, else None

  # Task 0 - Edge case - piece is Machine, it cannot have a right connection
  neighbour = item at (x,y) == (grid cell right of piece, yPieceSource)
  if piece is Machine and neighbour exists:
    yNeighbourSource = y-coordinate of Source of neighbour
    if yPieceSource == yNeighbourSource:
      call invalid position handler (also shows the error message)
      return

```

```
# Task 1 - deal with the connection at the Source interface of piece
neighbour = item at (x,y) == (grid cell left of piece, yPieceSource)
if neighbour exists:
  yNeighbourTarget = y-coordinate of Target of neighbour
  yNeighbourImplem = y-coordinate of Implem of neighbour

  if neighbour is Machine:
    # Case 1 - cannot connect Source of piece to Machine
    call invalid position handler (also shows the error message)
    return
  elif yNeighbourTarget is not None and yPieceSource == yNeighbourTarget:
    # Case 2 - Source of piece connected to Target of neighbour
    if Source language of piece == Target language of neighbour:
      # Target <- Source match
      disconnect old neighbour of Source (n.targetLink or n.implemLink = None)
      piece.sourceLink = neighbour
      neighbour.targetLink = piece
    else:
      # Positions of Target and Source match but languages mismatch
      call invalid position handler (also shows the error message)
      return
  elif yPieceSource == yNeighbourImplem:
    # Case 3 - Source of piece connected to Implem of neighbour
    same as Case 2 but every instance of Target is replaced by Implem
  else:
    # Case 4 - Source of piece not connected to Target or Implem of neighbour
    disconnect old neighbour of Source (n.targetLink or n.implemLink = None)
    piece.sourceLink = None
else:
  # Case 5 - Source of piece has no neighbour
  same as Case 4

# Task 2 - deal with the connection at the Target interface of piece
# Using yPieceSource instead of yPieceTarget in the next line avoids 1 if
neighbour = item at (x,y) == (2 grid cells right of piece, yPieceSource)
if yPieceTarget exists and neighbour exists:
  yNeighbourSource = y-coordinate of Source of neighbour

  if yPieceTarget == yNeighbourSource:
    # Case 1 - Target of piece connected to Source of neighbour
    if Target language of piece == Source language of neighbour:
      # Target -> Source match
      disconnect old neighbour of Target (n.sourceLink = None)
      piece.targetLink = neighbour
      neighbour.sourceLink = piece
    else:
      # Positions of Target and Source match but languages mismatch
      call invalid position handler (also shows the error message)
      return
  else:
    # Case 2 - Target of piece not connected to Source of neighbour
    disconnect old neighbour of Target (n.sourceLink = None)
    piece.targetLink = None
elif yPieceTarget:
  # Case 3 - Target of piece has no neighbour
  same as Case 2
# Case 4 - do nothing when piece has no Target interface
```

```
# Task 3 - deal with the connection at the Implem interface of piece
# No way of avoiding 1 if as in Task 2
if yPieceImplem:
  neighbour = item at (x,y) == (2 grid cells right of piece, yPieceImplem)
  if neighbour exists:
    yNeighbourSource = y-coordinate of Source of neighbour

    if yPieceImplem == yNeighbourSource:
      # Case 1 - Implem of piece connected to Source of neighbour
      if Implem language of piece == Source language of neighbour:
        # Implem -> Source match
        if piece.implemLink != piece.targetLink:
          # Edge case condition caused by the execution order, the case is
          # moving from p.Implem -> n.Source to p.Target -> n.Source
          disconnect old neighbour of Implem (n.sourceLink = None)
        piece.implemLink = neighbour
        neighbour.sourceLink = piece
      else:
        # Positions of Implem and Source match but languages mismatch
        if called from change height:
          call invalid height handler
        else:
          call invalid position handler (also shows the error message)
    else:
      # Case 2 - Implem of piece not connected to Source of neighbour
      if piece.implemLink != piece.targetLink:
        # Same edge case as in Case 1
        disconnect old neighbour of Implem (n.sourceLink = None)
      piece.implemLink = None
  else:
    # Case 3 - Implem of piece has no neighbour
    same as Case 2
# Case 4 - do nothing when piece has no Implem interface
```

Listing 5.1: *The connecting pieces check algorithm expressed as pseudocode.*

We could further describe how we implemented the invalid height and position handlers called inside the algorithm. However, that would mean going into very concrete implementation details. One more thing to mention is that the code of Task 3 is in fact written as a separate function. It is called *implemNeighbourCheck* and the reason for extracting the code was because the function is not only called in the algorithm but also in the functions *heightenPiece* and *shortenPiece*. This ensures changing height is only allowed when the connected languages match.

Since animation is implemented by copying the pieces from the diagram scene onto the animation scene, there is the *prepareAnimation* function. This function populates and returns two lists – one with the pieces stored in the animation order and the other with the corresponding text output. Each animation is prepared (and executed) as breadth-first traversal of the diagram starting at the piece selected by the user. Figure 5.1 demonstrates the animation order. We implemented this using a double-ended queue (*deque*). We start by appending the starting piece to the *deque* and keep traversing while the *deque* is not empty. At each iteration, we remove and return the left piece from the *deque*, append to the *deque* the piece's target and implementation neighbours if they exist, prepare the piece's text output and finally collect the piece and its text in the two lists.

The remaining functions are straightforward implementations of the scene-related actions such as deleting a piece (removing links is key here), clearing the scene, switching the grid and showing the appropriate dialog messages.

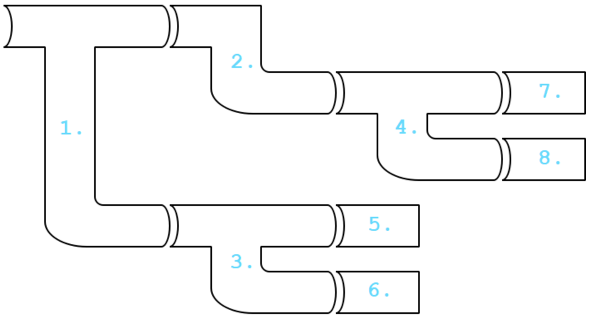


Figure 5.1: An artificial diagram with no languages (to simplify connecting pieces) demonstrating the animation order as breadth-first traversal of the diagram.

5.5 Animation window

AnimationWidget is the class containing the implementation of the animation window functionality. It has "Widget" in its title because it subclasses *QWidget*. In Qt, a widget which is not incorporated in another widget becomes a window. The initialisation of this class mostly consists of laying out the individual animation elements (Section 4.5) and setting up their properties. Other than the overridden event handler which ensures no references to the window are kept once it is closed (*closeEvent*), all functions in the class are either slots or utility functions called from within the slots. In Qt, slots are functions called as a result of an event emitting a signal. In this case, the event is the user pressing one of the play control buttons. There is a unique slot connected to each of the five buttons. The implementation of the slots uses a Boolean flag to execute actions conditionally depending on if the animation is playing or not. Highlighting and dehighlighting of the correct piece and showing the right text output is based on an integer instance variable used to index the list of pieces in the animation order. Both variables are updated in the slots such that the functionality is in line with the behaviour described in Section 4.5. When the animation is playing, the break between the steps is set to 1 second. The implementation of this uses the function *singleShot* from the *QTimer* class, not the *sleep* function of Python which would freeze the entire program because of suspending the event loop of Qt. Figure 5.2 shows the final implementation of the animation window.

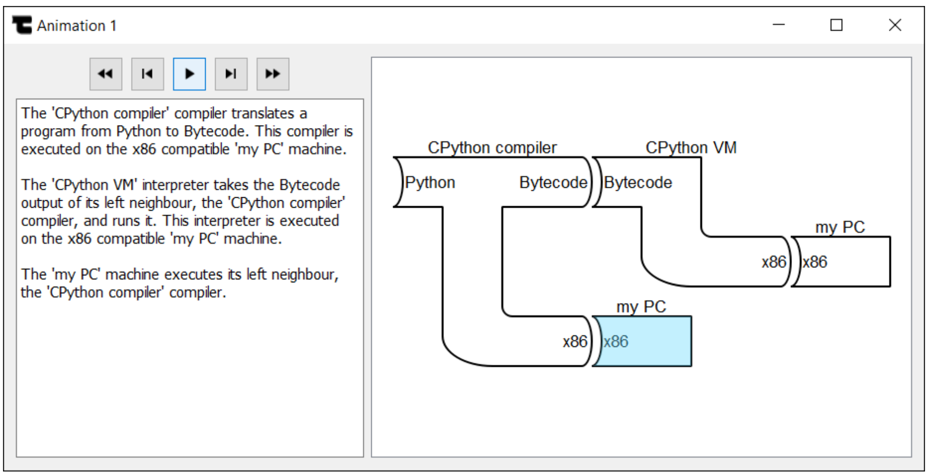


Figure 5.2: The final implementation of the animation window showing the execution of a Python program. Note that the animation is paused at the penultimate step.

5.6 Main window

Unsurprisingly, the main window is implemented in the *MainWindow* class which is a subclass of *QMainWindow*. The initialisation of course includes instantiating the diagram scene and the list of pieces and creating the menu bar, toolbar and actions which are triggered from these widgets or through the keyboard shortcuts.

We identified in Requirement 29 that the exported diagrams should use a common format which is human-readable and editable. To fulfil the requirement, our implementation exports the diagrams as JSON. The *exportDiagram* function exports every piece in the diagram scene. This is done by iterating over the scene pieces and in each iteration, appending a dictionary with the piece data to the list representing the full JSON. The user is then prompted for a file name which is where the JSON is dumped. An example of exported diagram file is shown in Listing A.2.

Each diagram we included in the *Examples* menu was created manually, then exported and set up to load after clicking on the corresponding menu action. In a sense, we bootstrapped our own examples! We included simple examples such as an interpretation and compilation of a program as well as more complex multistage diagrams involving bootstrapping. The list of all 15 examples is shown in Figure 5.3.

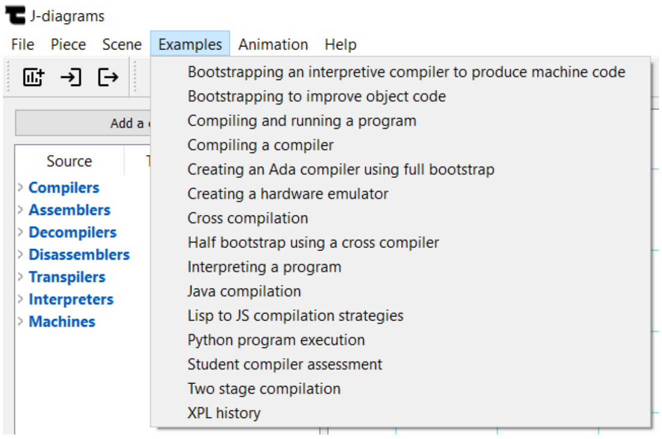


Figure 5.3: *The list of predefined example diagrams available in our software.*

When a diagram is imported directly or loaded from the *Examples* menu, our implementation uses the function *addPiecesFromJSON* which simply iterates over the JSON representation of the pieces, uses the data to create the corresponding *DiagramPiece* objects and adds the pieces to the diagram scene. Since the links between pieces are run-time references, they are not stored in the JSON. Because of this, it is crucial to call the *connecting pieces check* algorithm (Section 5.4) to establish the links and thus enable the animation to work correctly. This also ensures that any manual JSON edits are verified by the system.

In Requirement 26, we aimed to make the *Examples* menu extensible by the user. We accomplished this and all the user needs to do is place a previously exported diagram in the examples directory found in the main directory. It is necessary to restart the program for any new examples to appear in the menu. Sharing the entire examples directory with other users means having the ability to work easily with the same examples. Our implementation of this feature is surprisingly simple. When the program is launched, we get the path of the examples directory using the *pathlib* module of Python, iterate over the files within, store their paths and names in lists which are used when creating the actions of the menu bar that load the individual examples. This could be extended to allow refreshing of examples from within the menu so the user would not need to restart the program.

The *saveScene* function is also implemented in this class. Based on the user's choice, the function renders the diagram scene into an SVG or PNG image and saves it in a file. Our implementation saves the scene as the bounding rectangle of all pieces, not the entire available scene, and automatically removes the grid from the image. We could have implemented this function in the *DiagramScene* class but we chose to keep all input/output functions in the parent *MainWindow* class.

In Section 4.2, we described the design of adding user-defined pieces. It made sense to extend the implementation of this (Section 5.2) so that the pieces could be automatically saved and loaded, thus providing persistence across executions. When the user closes the main window, the *closeEvent* handler is called. In this function, using a Boolean flag, we check if the user added any custom pieces which we collect in the *PiecesTree* class in a dictionary of categories and lists of pieces. If so, the user is shown a dialog giving them an option to save the pieces. If they press Yes, we load the *custom_pieces* JSON file stored in the main directory, append to it the new custom pieces and save it. Hence, the file is a config file used for storing all added pieces. An example of the file is shown in Listing A.3. Our implementation choices allow the user to edit the file manually and share it with other users. To offer further flexibility, all custom pieces can be removed from the program by simply deleting the *custom_pieces* file. The default, empty version of the file is automatically created at launch if it does not exist. Implementing the functionality described in this paragraph was a realisation of Requirements 33 and 34. Note that the last step in the *closeEvent* function is unrelated to this functionality and its purpose is to close all animation windows before the main window is finally closed.

Figure 5.4 shows the final implementation of the main window with a multistage bootstrapping example loaded.

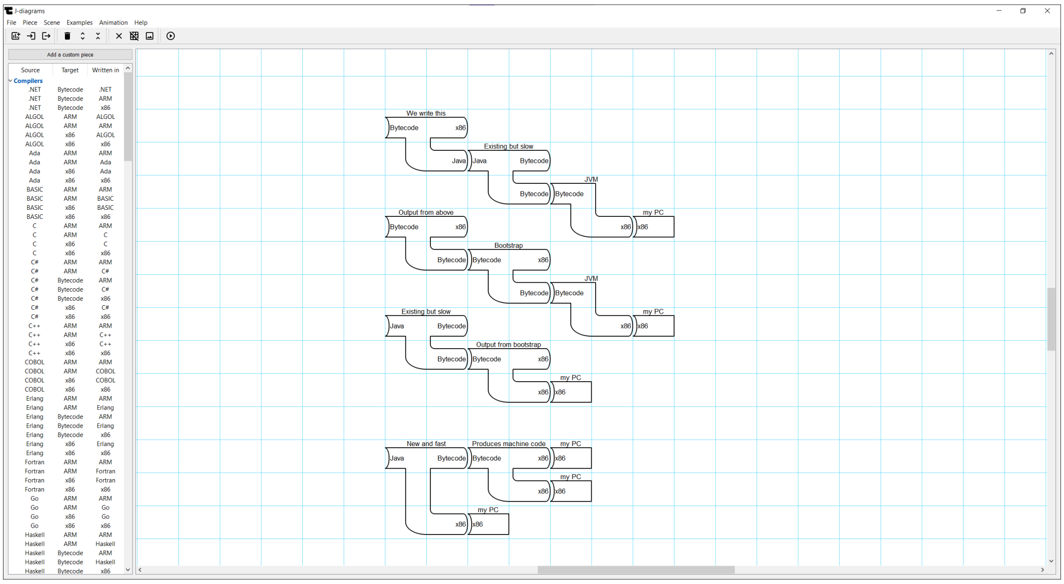


Figure 5.4: *The final implementation of the main window. The diagram illustrates bootstrapping an interpretive compiler to produce machine code.*

6 Evaluation

This chapter demonstrates how we evaluated our software using two independent evaluation methods. We first focus on our evaluation through usability heuristics and in the second section, we present the results of our user study. The first method was used whenever an individual feature was designed and implemented. The user study was conducted after fully implementing the software.

6.1 Evaluation through usability heuristics

In this method, we followed "10 general principles for interaction design" as introduced and updated by Nielsen (1994, 2020). We demonstrate how we transformed each principle into our software with one or more examples.

1. Visibility of system status
    - After adding a custom piece, feedback is given to the user by scrolling to the new piece and selecting it.
    - When a piece is clicked in the list of pieces, the piece is highlighted in light blue.
    - When a piece is placed such that it overlaps or has mismatched languages with another piece, the culprit is highlighted red.
    - When a diagram is loaded or imported, the view is focused on it. Without this, the user could be in a different part of the view and thus would not know the diagram appeared in the scene.

2. Match between system and the real world
    - In the list of pieces, all languages for the particular piece are presented together in one row and not in separate steps. This matches the essence of the corresponding language processor.
    - Using the word "pieces" for the elements of diagrams. This is an allusion to jigsaw puzzle pieces which are connected in a similar way.

3. User control and freedom
    - Every dialog gives the user an option to reject or cancel the interaction.
    - The ability to remove all custom pieces.

4. Consistency and standards
    - When a custom piece is added, it can be found in the same place in future launches.
    - Disallowed dragging of pieces within the list of pieces also ensures consistent positions.
    - All error messages follow a consistent format.
    - Keyboard shortcuts follow conventions, e.g. Ctrl+X for Exit, Ctrl+O for Import, F1 for Guide.

5. Error prevention
   - Invalid positioning and invalid height changes of pieces resulting from mismatched languages or overlapping pieces are disallowed.
   - Meaningless actions such as changing the height of machines or exporting an empty scene are disallowed.
   - When adding a custom piece, the language fields that are not needed are inactive.
   - The connecting pieces check algorithm is called when a diagram is loaded or imported to verify any manual edits to the diagram file.
   - Before committing to a destructive action such as wiping a diagram, a confirmation dialog is shown.

6. Recognition rather than recall
   - The use of icons chosen to represent the underlying actions meaningfully.

7. Flexibility and efficiency of use
   - The inclusion of toolbar and keyboard shortcuts in addition to the menu bar.

8. Aesthetic and minimalist design
   - Simple design which enables the functionality and does not distract with fancy graphics.
   - Toolbar excluding the rarely needed Help menu actions.
   - Diagramming available immediately after launching the program.

9. Help users recognize, diagnose, and recover from errors
   - No references to implementation/code details in the error messages.
   - The error messages consisting of an informative title, explanation why the error occurred and how it will be rectified.

10. Help and documentation
    - The inclusion of the video and text guide, both of which are presented in sections by using video chapters and text headings.

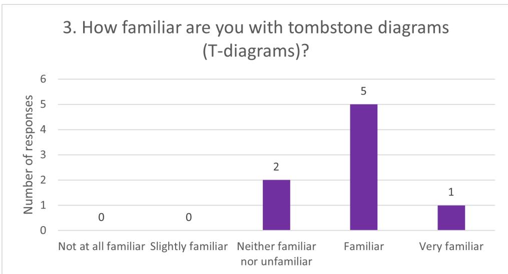
6.2 Evaluation through user study

In our user study, we collected qualitative data using a survey. The aim of the survey was to evaluate our software as a whole by asking the potential users for their opinion. Prior to conducting the study, we familiarised ourselves with the ethics checklist for the honours individual projects involving participants. We designed the study such that all conditions of the checklist were met. The signed ethics form is included as Appendix B.1. Our survey started with an introduction script where we stated the aim of the study and explained how it would proceed. Before continuing, the participants were asked to confirm they understood the introductory information and that they agreed to take part. We embedded a video in the survey in which we demonstrated our software. The participants were asked to watch the video and answer several questions in the Likert scale, yes-no, or text input format. We concluded the survey with a pair of demographic questions and a debriefing script. The entire survey is included as Appendix B.2. We distributed our survey to the Year 3 and 4 Computing Science students at the University of Glasgow. We managed to collect responses from eight participants, seven of which have taken the Programming Languages course. The results were as follows.

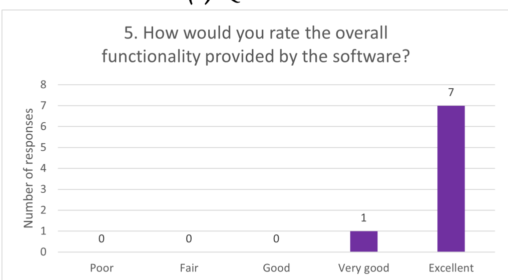
The level of familiarity with tombstone diagrams was above average with five participants choosing the answer "familiar" on a five-point Likert scale. One participant was "very familiar" and two chose the neutral option. This distribution is visualised in Figure 6.1a. It was very encouraging to find that all participants thought that animation is a useful educational tool. This justified the point of our project. These two questions were asked before the video demonstration. It was also encouraging to find that the functionality provided by our software was rated as "excellent". Seven participants chose this option. The last participant answered "very good" as visualised in Figure 6.1b. We extended this question by asking the participant what they would change about the functionality. Three participants would not change anything, one participant would add the possibility to toggle the grid on and off which is already possible. The remaining participants provided the following comments.

1. Incorporate some gamification and tutorials in which users would have to complete the holes in diagrams to learn about J-diagrams and processing pipelines.
2. The ability to continue an animation by transferring a state (as was done manually in the last diagram).
3. Filter pieces by language/architecture.
4. Check for invalid placement before prompting to enter name when inserting a new piece - fail early.
5. The pop-ups displaying errors might become annoying. Highlighting the object red and marking it with a label explaining the invalid operation might be more user friendly. The object could then revert back to normal once the user is no longer performing an invalid operation with it.

The comments 1-4 are excellent suggestions that we would take further as future work. The first part of the comment 5 is something that did occur to us. The logic behind our design was that we needed to give the user explicit feedback so that they could learn. The suggested alternative could be implemented such that the label is an exclamation mark icon placed in the middle of a piece and hovering over it would display the error. This is certainly something worth looking into but would require some time since it would require numerous changes to the existing code.



(a) Question 3



(b) Question 5

Figure 6.1: *The distribution of answers to Question 3 and 5 in our user study.*

Our design was not rated as highly as the functionality. Four participants rated the design as "excellent", three as "very good" and one as "good". Figure 6.2 visualises this distribution. One participant said the designed looked "fine for its purpose" but "designed in 2000-2010". While we acknowledge that our design may not be award-winning, the point was not to create the most beautiful product to attract customers. The point was to create a design that would support the functionality. Additionally, a more complex design would go against the philosophy of minimalist design we chose to follow (Usability Heuristic 8). Four participants provided concrete suggestions how the design could be changed. One person would like to be able to choose the colour of pieces "for visual clarity in larger diagrams". This is something which could be implemented easily. However, there would be a danger of clashing colours when highlighting invalid positions or animating. The remaining comments were about a different design for the list of pieces. The consensus was that there should be more filtering options which we agree with. One suggestions was to have a design similar to the alternative presented in the last paragraph of Section 4.1.

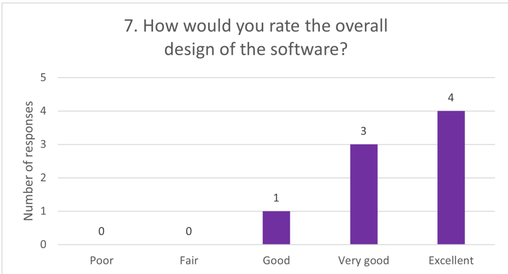


Figure 6.2: *The distribution of answers to Question 7 in our user study.*

After the questions about the functionality and design, we showed the participants the list of predefined examples included in the program and asked them if they thought the examples demonstrated a varied range of concepts. All but one participant answered yes.

We recorded the demonstration video included in the survey such that it would be used as the software guide. Hence, we asked if the participants considered the video format of the guide appropriate. Five participants were happy with the video. The rest would prefer to have both video and text guide. Both formats are included in the program and the answers confirm our decision was correct. The guides are attached in Appendix C.

Since our software uses a new design of tombstone diagrams, we wanted to know how the participants felt about the J-diagrams notation. Six participants felt that J-diagrams were straightforward and an improvement on T-diagrams. This is very encouraging and reaffirms the point we made in Section 2.2 about the ease of transitioning from T-diagrams to J-diagrams. One participant said they were unfamiliar with J-diagrams which is not surprising considering the notation is less than a year old. The participant would like to look into J-diagrams. However, they also suggested the feature of switching between the two notations. This is not a good idea because it would mean going back to the problems of T-diagrams. If there were serious problems with J-diagrams, e.g. the users disliking the fact that the output is implicit, they should be rectified in that notation. The last participant found J-diagrams more confusing which would be interesting to investigate further.

The responses to the final question revealed that seven participants would use the program if it were incorporated into the Programming Languages course. However, two answers were conditioned on the amount of content related to J-diagrams. The last participant would "maybe" use the program "a bit" and thought the diagramming was sufficient enough and the animation did not add much. This could be because the participant already understood the concepts and would only benefit from using the program as a revision tool.

7 Conclusion

7.1 Summary

The goal of our project was to create a piece of software that would enable the user to create and animate arbitrary tombstone diagrams.

In Chapter 2, after introducing the concepts relevant to the diagrams, we demonstrated how the J-diagrams notation improved the traditional T-diagrams. Hence, our choice to use the improved notation in our software. We explored animation as an educational tool and found that it was effective. In our related software research, we found there was no software for animating tombstone diagrams, hence validating our project.

In Chapter 3, we presented the concrete individual requirements for the project which were categorised using the MoSCoW prioritisation technique. We also identified four types of users of our software and summarised them through user stories.

In Chapter 4 and 5, we described how we designed and implemented the individual elements of the software and how we combined them into the overall product. We met the project requirements by implementing all “Must have” and “Should have” requirements listed in Section 3.1. We also implemented Requirement 37 from the “Could have” category. This corresponded to 35 out of 50 requirements.

In Chapter 6, we demonstrated how we evaluated our software through usability heuristics and a user study. We found that all participants thought animation was a useful educational tool. The functionality of our software was rated on average at 4.88 out of 5 and the design at 4.38 out of 5. The participants made useful suggestions for future work. The key focus should be on improving the list of pieces with filtering options and making it possible to animate multistage diagrams in a single animation window. A key finding was that the majority of participants found J-diagrams straightforward. Hence, we would suggest that they are used instead of T-diagrams in the courses involving language processors. The majority of the participants would also use our software if it were a part of the Programming Languages course at the University of Glasgow. Our software has not been distributed yet but we will make it available to the students of the course as well as the computing science public after May 2021.

7.2 Reflection

This project was the biggest individual piece of work we have ever undertaken. Thanks to a thorough background research, we have improved our knowledge of language processors. The development of our software included a mixture of user interface design, program structure design, algorithm creation and writing the corresponding code. All of these are invaluable skills. We have improved not only in Qt/PySide2 programming but also in studying official documentation which was necessary throughout the implementation process. Conducting a user evaluation, thanks to which we obtained useful comments, reminded us about the importance of that step. Other than creating a useful program as a whole, the biggest individual achievement was the implementation of the *connecting pieces check* algorithm.

Although we can say the project was a success, there are things we wish we could have done better. All our testing was done as usability testing. It would be beneficial to have some automatic tests as well. However, this is always a challenge when testing something as interactive as a graphical user interface. Despite the project not being classified as a software engineering project, we tried to follow software engineering best practices as much as possible. We wrote self-documenting code through appropriate naming but we were unable to include Python docstrings in every function due to the sheer number of functions. This is a non-functional requirement for future work. Unfortunately, our skills prior to the project did not allow us to implement the program directly with Qt in C++. Functionality-wise, there would have been no difference. There would not have been much of a performance difference either. However, developing in C++ would contribute immensely to our personal development. Finally, we would have liked to conduct the user study in person and with more participants. This was not possible due to the COVID-19 restrictions.

7.3 Future work

The first group of suggestions for future work is the unimplemented “Could have” and “Won't have this time” requirements from Section 3.1 except those we explicitly rejected in Chapter 4. This would mean implementing Requirements 35, 36, 38–40, 42–44 and 46–49. The second group is the suggestions 1-5 made by the user study participants. Other suggestions we make are as follows.

- A search bar at the top of the list of pieces.
- Allow the user, e.g. a lecturer, to define the animation text.
- An option to insert text to the diagram scene. This could be implemented as a free form text box. It would not snap to the grid and could be positioned freely anywhere near the pieces. It would be useful for labelling pieces and providing descriptions in a more flexible way than only through the names.
- Deleting pieces from the list of pieces using a keyboard or mouse right-click.
- Optimisations to the JSON files. Rather than storing “-” for every interpreter and machine, the files could be changed to store only the required languages and add “-” programmatically. This would save storage. On the other hand, this would not correspond to the representation of the pieces and the file size is unlikely to be too large in the first place.

A | Code and Data Listings

```
{
  "Compilers": [
     (".NET", "Bytecode", ".NET"),
     (".NET", "Bytecode", "ARM"),
     (".NET", "Bytecode", "x86"),
     ("ALGOL", "ARM", "ALGOL"),
     ("ALGOL", "ARM", "ARM"),
     ("ALGOL", "x86", "ALGOL"),
     ("ALGOL", "x86", "x86"),
     ("Ada", "ARM", "ARM"),
     ("Ada", "ARM", "Ada"),
     ("Ada", "x86", "Ada"),
     ("Ada", "x86", "x86"),
     ("BASIC", "ARM", "ARM"),
     ("BASIC", "ARM", "BASIC"),
     ("BASIC", "x86", "BASIC"),
     ("BASIC", "x86", "x86"),
     ("C", "ARM", "ARM"),
     ("C", "ARM", "C"),
     ("C", "x86", "C"),
     ("C", "x86", "x86"),
     ("C#", "ARM", "ARM"),
     ("C#", "ARM", "C#"),
     ("C#", "Bytecode", "ARM"),
     ("C#", "Bytecode", "C#"),
     ("C#", "Bytecode", "x86"),
     ("C#", "x86", "C#"),
     ("C#", "x86", "x86"),
     ("C ", "ARM", "ARM"),
     ("C ", "ARM", "C "),
     ("C ", "x86", "C "),
     ("C ", "x86", "x86"),
     ("COBOL", "ARM", "ARM"),
     ("COBOL", "ARM", "COBOL"),
     ("COBOL", "x86", "COBOL"),
     ("COBOL", "x86", "x86"),
     ("Erlang", "ARM", "ARM"),
     ("Erlang", "ARM", "Erlang"),
     ("Erlang", "Bytecode", "ARM"),
     ("Erlang", "Bytecode", "Erlang"),
     ("Erlang", "Bytecode", "x86"),
     ("Erlang", "x86", "Erlang"),
     ("Erlang", "x86", "x86"),
     ("Fortran", "ARM", "ARM"),
     ("Fortran", "ARM", "Fortran"),
     ("Fortran", "x86", "Fortran"),
```

```
    ("Fortran", "x86", "x86"),
    ("Go", "ARM", "ARM"),
    ("Go", "ARM", "Go"),
    ("Go", "x86", "Go"),
    ("Go", "x86", "x86"),
    ("Haskell", "ARM", "ARM"),
    ("Haskell", "ARM", "Haskell"),
    ("Haskell", "Bytecode", "ARM"),
    ("Haskell", "Bytecode", "Haskell"),
    ("Haskell", "Bytecode", "x86"),
    ("Haskell", "x86", "Haskell"),
    ("Haskell", "x86", "x86"),
    ("Java", "ARM", "ARM"),
    ("Java", "ARM", "Java"),
    ("Java", "Bytecode", "ARM"),
    ("Java", "Bytecode", "Java"),
    ("Java", "Bytecode", "x86"),
    ("Java", "x86", "Java"),
    ("Java", "x86", "x86"),
    ("Lisp", "ARM", "ARM"),
    ("Lisp", "ARM", "Lisp"),
    ("Lisp", "Bytecode", "ARM"),
    ("Lisp", "Bytecode", "Lisp"),
    ("Lisp", "Bytecode", "x86"),
    ("Lisp", "x86", "Lisp"),
    ("Lisp", "x86", "x86"),
    ("Lua", "Bytecode", "ARM"),
    ("Lua", "Bytecode", "Lua"),
    ("Lua", "Bytecode", "x86"),
    ("ML", "ARM", "ARM"),
    ("ML", "ARM", "ML"),
    ("ML", "x86", "ML"),
    ("ML", "x86", "x86"),
    ("ObjectiveC", "ARM", "ARM"),
    ("ObjectiveC", "ARM", "ObjectiveC"),
    ("ObjectiveC", "x86", "ObjectiveC"),
    ("ObjectiveC", "x86", "x86"),
    ("Pascal", "ARM", "ARM"),
    ("Pascal", "ARM", "Pascal"),
    ("Pascal", "x86", "Pascal"),
    ("Pascal", "x86", "x86"),
    ("Python", "Bytecode", "ARM"),
    ("Python", "Bytecode", "Python"),
    ("Python", "Bytecode", "x86"),
    ("Rust", "ARM", "ARM"),
    ("Rust", "ARM", "Rust"),
    ("Rust", "x86", "Rust"),
    ("Rust", "x86", "x86"),
    ("Scala", "ARM", "ARM"),
    ("Scala", "ARM", "Scala"),
    ("Scala", "Bytecode", "ARM"),
    ("Scala", "Bytecode", "Scala"),
    ("Scala", "Bytecode", "x86"),
    ("Scala", "x86", "Scala"),
    ("Scala", "x86", "x86"),
    ("Swift", "ARM", "ARM"),
    ("Swift", "ARM", "Swift"),
```

```
        ("Swift", "x86", "Swift"),
        ("Swift", "x86", "x86"),
    ],
    "Assemblers": [
        ("ARM asm", "ARM", "ARM"),
        ("MIPS asm", "MIPS", "MIPS"),
        ("PPC asm", "PPC", "PPC"),
        ("SPARC asm", "SPARC", "SPARC"),
        ("x86 asm", "x86", "x86"),
    ],
    "Decompilers": [
        ("ARM", "ALGOL", "ALGOL"),
        ("ARM", "ALGOL", "ARM"),
        ("ARM", "Ada", "ARM"),
        ("ARM", "Ada", "Ada"),
        ("ARM", "BASIC", "ARM"),
        ("ARM", "BASIC", "BASIC"),
        ("ARM", "C", "ARM"),
        ("ARM", "C", "C"),
        ("ARM", "C#", "ARM"),
        ("ARM", "C#", "C#"),
        ("ARM", "C ", "ARM"),
        ("ARM", "C ", "C "),
        ("ARM", "COBOL", "ARM"),
        ("ARM", "COBOL", "COBOL"),
        ("ARM", "Erlang", "ARM"),
        ("ARM", "Erlang", "Erlang"),
        ("ARM", "Fortran", "ARM"),
        ("ARM", "Fortran", "Fortran"),
        ("ARM", "Go", "ARM"),
        ("ARM", "Go", "Go"),
        ("ARM", "Haskell", "ARM"),
        ("ARM", "Haskell", "Haskell"),
        ("ARM", "Java", "ARM"),
        ("ARM", "Java", "Java"),
        ("ARM", "Lisp", "ARM"),
        ("ARM", "Lisp", "Lisp"),
        ("ARM", "ML", "ARM"),
        ("ARM", "ML", "ML"),
        ("ARM", "ObjectiveC", "ARM"),
        ("ARM", "ObjectiveC", "ObjectiveC"),
        ("ARM", "Pascal", "ARM"),
        ("ARM", "Pascal", "Pascal"),
        ("ARM", "Rust", "ARM"),
        ("ARM", "Rust", "Rust"),
        ("ARM", "Scala", "ARM"),
        ("ARM", "Scala", "Scala"),
        ("ARM", "Swift", "ARM"),
        ("ARM", "Swift", "Swift"),
        ("Bytecode", ".NET", ".NET"),
        ("Bytecode", ".NET", "ARM"),
        ("Bytecode", ".NET", "x86"),
        ("Bytecode", "C#", "ARM"),
        ("Bytecode", "C#", "C#"),
        ("Bytecode", "C#", "x86"),
        ("Bytecode", "Erlang", "ARM"),
        ("Bytecode", "Erlang", "Erlang"),
```

```
    ("Bytecode", "Erlang", "x86"),
    ("Bytecode", "Haskell", "ARM"),
    ("Bytecode", "Haskell", "Haskell"),
    ("Bytecode", "Haskell", "x86"),
    ("Bytecode", "Java", "ARM"),
    ("Bytecode", "Java", "Java"),
    ("Bytecode", "Java", "x86"),
    ("Bytecode", "Lisp", "ARM"),
    ("Bytecode", "Lisp", "Lisp"),
    ("Bytecode", "Lisp", "x86"),
    ("Bytecode", "Lua", "ARM"),
    ("Bytecode", "Lua", "Lua"),
    ("Bytecode", "Lua", "x86"),
    ("Bytecode", "Python", "ARM"),
    ("Bytecode", "Python", "Python"),
    ("Bytecode", "Python", "x86"),
    ("Bytecode", "Scala", "ARM"),
    ("Bytecode", "Scala", "Scala"),
    ("Bytecode", "Scala", "x86"),
    ("x86", "ALGOL", "ALGOL"),
    ("x86", "ALGOL", "x86"),
    ("x86", "Ada", "Ada"),
    ("x86", "Ada", "x86"),
    ("x86", "BASIC", "BASIC"),
    ("x86", "BASIC", "x86"),
    ("x86", "C", "C"),
    ("x86", "C", "x86"),
    ("x86", "C#", "C#"),
    ("x86", "C#", "x86"),
    ("x86", "C ", "C "),
    ("x86", "C ", "x86"),
    ("x86", "COBOL", "COBOL"),
    ("x86", "COBOL", "x86"),
    ("x86", "Erlang", "Erlang"),
    ("x86", "Erlang", "x86"),
    ("x86", "Fortran", "Fortran"),
    ("x86", "Fortran", "x86"),
    ("x86", "Go", "Go"),
    ("x86", "Go", "x86"),
    ("x86", "Haskell", "Haskell"),
    ("x86", "Haskell", "x86"),
    ("x86", "Java", "Java"),
    ("x86", "Java", "x86"),
    ("x86", "Lisp", "Lisp"),
    ("x86", "Lisp", "x86"),
    ("x86", "ML", "ML"),
    ("x86", "ML", "x86"),
    ("x86", "ObjectiveC", "ObjectiveC"),
    ("x86", "ObjectiveC", "x86"),
    ("x86", "Pascal", "Pascal"),
    ("x86", "Pascal", "x86"),
    ("x86", "Rust", "Rust"),
    ("x86", "Rust", "x86"),
    ("x86", "Scala", "Scala"),
    ("x86", "Scala", "x86"),
    ("x86", "Swift", "Swift"),
    ("x86", "Swift", "x86"),
```

```
    ],
    "Disassemblers": [
        ("ARM", "ARM asm", "ARM"),
        ("MIPS", "MIPS asm", "MIPS"),
        ("PPC", "PPC asm", "PPC"),
        ("SPARC", "SPARC asm", "SPARC"),
        ("x86", "x86 asm", "x86"),
    ],
    "Transpilers": [
        (".NET", "JavaScript", "ARM"),
        (".NET", "JavaScript", "x86"),
        ("C", "JavaScript", "ARM"),
        ("C", "JavaScript", "x86"),
        ("C", "Rust", "ARM"),
        ("C", "Rust", "x86"),
        ("C#", "JavaScript", "ARM"),
        ("C#", "JavaScript", "x86"),
        ("C ", "C", "ARM"),
        ("C ", "C", "x86"),
        ("C ", "JavaScript", "ARM"),
        ("C ", "JavaScript", "x86"),
        ("CoffeeScript", "JavaScript", "ARM"),
        ("CoffeeScript", "JavaScript", "x86"),
        ("Go", "JavaScript", "ARM"),
        ("Go", "JavaScript", "x86"),
        ("Haskell", "C", "ARM"),
        ("Haskell", "C", "x86"),
        ("Java", "JavaScript", "ARM"),
        ("Java", "JavaScript", "x86"),
        ("Java", "Objective-C", "ARM"),
        ("Java", "Objective-C", "x86"),
        ("Lua", "JavaScript", "ARM"),
        ("Lua", "JavaScript", "x86"),
        ("Objective-C", "Swift", "ARM"),
        ("Objective-C", "Swift", "x86"),
        ("Perl", "JavaScript", "ARM"),
        ("Perl", "JavaScript", "x86"),
        ("Python", "Java", "ARM"),
        ("Python", "Java", "x86"),
        ("Python", "JavaScript", "ARM"),
        ("Python", "JavaScript", "x86"),
        ("Python2", "Python3", "ARM"),
        ("Python2", "Python3", "x86"),
        ("Ruby", "JavaScript", "ARM"),
        ("Ruby", "JavaScript", "x86"),
        ("Rust", "JavaScript", "ARM"),
        ("Rust", "JavaScript", "x86"),
        ("Scala", "JavaScript", "ARM"),
        ("Scala", "JavaScript", "x86"),
        ("TypeScript", "JavaScript", "ARM"),
        ("TypeScript", "JavaScript", "x86"),
    ],
    "Interpreters": [
        ("BASIC", "-", "ARM"),
        ("BASIC", "-", "x86"),
        ("Bytecode", "-", "ARM"),
        ("Bytecode", "-", "C"),
```

```
        ("Bytecode", "-", "C "),
        ("Bytecode", "-", "x86"),
        ("Haskell", "-", "ARM"),
        ("Haskell", "-", "x86"),
        ("Java", "-", "ARM"),
        ("Java", "-", "x86"),
        ("JavaScript", "-", "ARM"),
        ("JavaScript", "-", "x86"),
        ("Lisp", "-", "ARM"),
        ("Lisp", "-", "x86"),
        ("Lua", "-", "ARM"),
        ("Lua", "-", "x86"),
        ("MATLAB", "-", "ARM"),
        ("MATLAB", "-", "x86"),
        ("OCaml", "-", "ARM"),
        ("OCaml", "-", "x86"),
        ("Perl", "-", "ARM"),
        ("Perl", "-", "x86"),
        ("Python", "-", "ARM"),
        ("Python", "-", "x86"),
        ("R", "-", "ARM"),
        ("R", "-", "x86"),
        ("Ruby", "-", "ARM"),
        ("Ruby", "-", "x86"),
    ],
    "Machines": [
        ("ARM", "-", "-"),
        ("MIPS", "-", "-"),
        ("PPC", "-", "-"),
        ("SPARC", "-", "-"),
        ("x86", "-", "-"),
    ],
}
```

Listing A.1: *The dictionary storing all default pieces. It was created through an automated approach as described in Section 5.2. The order in the tuples is: source, target, implementation language.*

```
[
  {
    "x": 4600.0,
    "y": 4800.0,
    "type": 2,
    "category": "machine",
    "name": "my PC",
    "languages": ["x86", "-", "-"],
    "height": 1
  },
  {
    "x": 4800.0,
    "y": 4720.0,
    "type": 2,
    "category": "machine",
    "name": "my PC",
    "languages": ["x86", "-", "-"],
    "height": 1
  },
  {
    "x": 4600.0,
    "y": 4640.0,
    "type": 1,
    "category": "interpreter",
    "name": "CPython VM",
    "languages": ["Bytecode", "-", "x86"],
    "height": 1
  },
  {
    "x": 4400.0,
    "y": 4640.0,
    "type": 0,
    "category": "compiler",
    "name": "CPython compiler",
    "languages": ["Python", "Bytecode", "x86"],
    "height": 2
  }
]
```

Listing A.2: *JSON representation of an exported diagram. The diagram is the one shown in Figure 5.2.*

```
{
  "Compilers": [ ["OCaml", "x86", "C"], ["Java 11", "Bytecode", "Java 10"] ],
  "Assemblers": [ ["Itanium asm", "Itanium", "Itanium"] ],
  "Decompilers": [],
  "Disassemblers": [],
  "Transpilers": [ ["Haskell", "C", "Java"] ],
  "Interpreters": [ ["Python", "-", "C"] ],
  "Machines": [ ["Itanium", "-", "-"] ]
}
```

Listing A.3: *An example of user-defined pieces stored in the custom_pieces JSON file.*

B User study documents

B.1 Ethics form

School of Computing Science
University of Glasgow

Ethics checklist form for 3rd/4th/5th year, and taught MSc projects

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.
   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.
   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.
   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.
   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.

12. All the data collected from the participants is stored in an anonymous form.
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title: Animation of tombstone diagrams

Student's Name: Michal Broos

Student Number: 2330994B

Student's Signature: Signed digitally

Supervisor's Name: Dr Ornela Dardha

Date: 02/04/2021

B.2 Survey

Animation of tombstone diagrams - Honours Individual Project Evaluation

* Required

Introduction

The aim of this survey is to evaluate a piece of software (referred to as "the software" in the rest of the survey) designed and implemented by Michal Broos as part of the honours individual project at the University of Glasgow. The software allows the user to create and animate arbitrary tombstone diagrams composed of programming language processors (translators, interpreters and machines).

It is impossible to tell how good the software is without asking the potential users, e.g. Programming Languages (H) students, for their opinion. Hence the need for this survey. You will be asked to watch a video demonstration of the software and answer a few questions. Your responses will be collected and stored anonymously. Please remember that it is the software, not you, that is being evaluated. You are welcome to withdraw at any time. If you do so, then it will not be possible for you to be debriefed.

If you have any questions, please do not hesitate to contact
Michal Broos at 2330994B@student.gla.ac.uk or Dr Ornela Dardha at Ornela.Dardha@glasgow.ac.uk

1. Before proceeding, you need to confirm that *

   Check all that apply.

   - [ ] You have read and understood the information in the Introduction section
   - [ ] You agree to take part and that your responses can be used in the project
   - [ ] You are at least 16 years old
   - [ ] You have no impairment that may limit your understanding or communication

2. When did you take the Programming Languages (H) course? *

   Mark only one oval.

   - ( ) 2021
   - ( ) 2020
   - ( ) Before 2020
   - ( ) I have not taken the course

3. How familiar are you with tombstone diagrams (T-diagrams)? *

Mark only one oval.

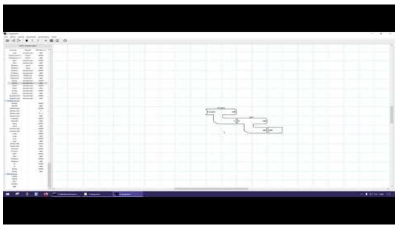
	1	2	3	4	5	
Not at all familiar	◯	◯	◯	◯	◯	Very familiar

4. Do you think that animation is a useful educational tool? *

Mark only one oval.

◯ Yes
◯ No

Before proceeding, please watch the following video which demonstrates the software. Please watch the video in full screen and the highest possible quality (1440p60 available).


http://youtube.com/watch?v=_Oj6TEqbuek

5. How would you rate the overall functionality provided by the software? *

Mark only one oval.

	1	2	3	4	5	
Poor	◯	◯	◯	◯	◯	Excellent

6. What, if anything, would you change about the functionality? *

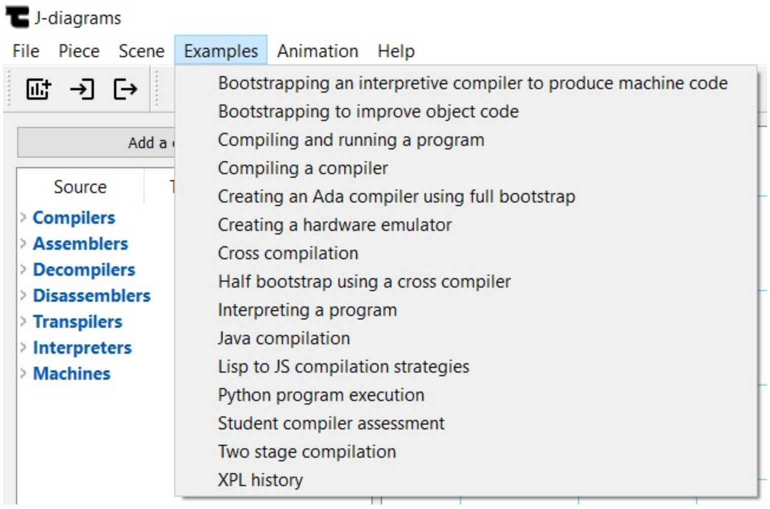
7. How would you rate the overall design of the software? *

Mark only one oval.

	1	2	3	4	5	
Poor	⬭	⬭	⬭	⬭	⬭	Excellent

8. What, if anything, would you change about the design? *

Please look at the examples included in the software and answer the question below the image.



9. Do you think the examples demonstrate a varied range of concepts involving programming language processors? *

Mark only one oval.

- Yes
- No

10. The video you watched serves as the software guide. Do you think that video is an appropriate format for the guide? If not, why and what format would you prefer? *

11. The software uses the redesigned tombstone diagrams, called J-diagrams, proposed by Wickerson and Brunet after identifying problems with the traditional T-diagrams. How did you feel about the J-diagrams notation? *

12. Would you use the software, if it were incorporated into the Programming Languages (H) course? If not, why? *

13. Please enter any other comments you might have.

Demographics

14. What is your age group? *

Mark only one oval.

- Prefer not to say
- 16-20
- 21-29
- 30-39
- 40-49
- 50 and over

15. What gender do you identify as? *

Mark only one oval.

- Prefer not to say
- Female
- Male
- Other: ____________

Debriefing

The aim of this survey was to evaluate the software for creating and animating tombstone diagrams.

If you have any further questions or comments, please see the contact details in the Introduction section.

Thank you very much for your participation.

C Guides

C.1 Video guide

The video guide is available at https://www.youtube.com/watch?v=_Oj6TEqbuek

C.2 Text guide

When you open the software, you are met with the screen which allows you to start creating diagrams straight away.

List of pieces

On the left, you can see the list of draggable pieces. These are divided into categories which can be collapsed. The categories include translators which are divided further, interpreters and machines. Each category includes numerous pieces.

Custom pieces

You can add a custom piece, meaning custom languages, not shapes, by clicking on the ***Add a custom piece*** button located above the **List of pieces**. Simply choose a category and enter your desired languages in the displayed dialog. After adding a piece, the software scrolls to it and highlights it. All custom pieces are added below the default pieces. If you close the software after adding pieces, you are given an option to keep them available in your future sessions. All added pieces are stored in the ***custom_pieces*** JSON file present in the main directory. This file can be edited following the existing structure, it can be shared between users and if you ever want to remove all custom pieces, simply delete the file and an empty one will be created on the next launch.

Diagramming

The pieces can be dragged from the list of pieces and dropped onto the diagram scene. After dropping a piece, you can name it in the displayed dialog or you can leave the name blank. The pieces snap to the grid and their snapping is based on the top left corner of the bounding rectangle. They can be connected from either side, but only if a connection is valid. If the connection is invalid, the software informs you about this and moves the piece back to its previous position. If a piece is dragged from the list of pieces into an invalid connection, the piece is deleted. Overlapping pieces are not allowed and are rectified in the same way.

Animating

Every diagram can be animated. Before animating, you need to select a starting piece. Confirming the prompt opens a new animation window with your selected diagram. Just like the main window, the animation window can be dynamically resized. Five play controls are available. You can play the entire animation and observe highlighting of the pieces and their corresponding textual output. You can pause and resume as you like. You can go forward and backward by one step. Or you can go directly to the beginning or the end. There is an unlimited number of animation windows you can open so you can animate multiple stages next to each other.

The text guide continues on the next page.

Menu bar + toolbar + keyboard shortcuts

Other than drag-and-drop, all available actions are accessible from the menu bar at the top. There is also a toolbar offering quick access to actions. Tooltips are included. Keyboard shortcuts are shown next to the actions in the menus.

File menu

In the File menu, you can create a new diagram. If there is a diagram already, you are asked to confirm that you are happy to wipe it. You can choose a blank diagram or a predefined example. You can import a diagram (the same confirmation prompt applies) and you can export a diagram. The file format is JSON which means you can edit it and most importantly share diagrams easily between users.

Piece menu

In the Piece menu, you can delete a selected piece and increase or decrease its height. This menu is also accessible by right clicking a piece. Changing height comes handy when creating more complex diagrams. Changing height into an invalid connection is not allowed and reverts the piece back to the previous height. Similarly, heightening into an overlap is impossible.

Scene menu

In the Scene menu, you can clear the diagram scene. Again, the confirmation prompt ensures you do not wipe a diagram by mistake. You can remove the grid and add it back. The grid is especially helpful when positioning pieces. You can also save the scene as image. This supports saving as SVG and PNG. The scene saved is the bounding rectangle of all pieces. Also, the grid is automatically removed in the image.

Examples menu

The Examples menu allows you to quickly open one of the predefined examples. This list can easily be extended by placing an exported diagram in the examples directory found within the main directory. You need to restart the software for any changes to take place. Sharing the entire examples directory with other users means you can all have the same Examples menu.

Animation menu

The only option available in this menu is to start animating a diagram as described already.

Help menu

This menu includes the Guide, Contact and About actions which are self-explanatory.

Exiting

Closing the main window closes all animation windows as well.

Bibliography

Agile Business Consortium (2021), 'Chapter 10: MoSCoW Prioritisation', `https://www.agilebusiness.org/page/ProjectFramework_10_MoSCoWPrioritisation`. Last accessed: 2021-02-27.

Bratman, H. (1961), 'An Alternate Form of the "UNCOL Diagram"', *Commun. ACM* **4**(3), 142.
URL: *https://doi.org/10.1145/366199.366249*

Burkhardt, W. H. (1965), Universal Programming Languages and Processors: A Brief Survey and New Concepts, *in* 'Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I', AFIPS '65 (Fall, part I), Association for Computing Machinery, New York, NY, USA, p. 1–21.
URL: *https://doi.org/10.1145/1463891.1463893*

Earley, J. and Sturgis, H. (1970), 'A Formalism for Translator Interactions', *Commun. ACM* **13**(10), 607–617.
URL: *https://doi.org/10.1145/355598.362740*

Fleischer, R. and Kučera, L. (2002), Algorithm Animation for Teaching, *in* 'Software Visualization', Springer Berlin Heidelberg, pp. 113–128.
URL: *https://link.springer.com/chapter/10.1007/3-540-45875-1_9*

Galles, D. (2011), 'Data Structure Visualizations', `https://www.cs.usfca.edu/~galles/visualization/Algorithms.html`. Last accessed: 2021-03-27.

Halim, S. (2011), 'VisuAlgo - visualising data structures and algorithms through animation', `https://visualgo.net/en`. Last accessed: 2021-03-27.

Hielscher, M. (2006*a*), 'AtoCC', `http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=en&site=main`. Last accessed: 2021-04-06.

Hielscher, M. (2006*b*), 'AtoCC - eine Lernumgebung für theoretische Informatik', `http://www.atocc.de/cgi-bin/atocc/site.cgi?lang=en&site=papers`. Last accessed: 2021-04-06.

Hundhausen, C. D., Douglas, S. A. and Stasko, J. T. (2002), 'A Meta-Study of Algorithm Visualization Effectiveness', *Journal of Visual Languages & Computing* **13**(3), 259–290.
URL: *https://www.sciencedirect.com/science/article/pii/S1045926X02902375*

Jakobsen, M. (2017), 'Tombstone diagram drawing package for LaTeX', `https://github.com/hrjakobsen/tombstone`. Last accessed: 2021-04-06.

Lagerstrom, L. (2003), *Programming the Web Using XHTML and JavaScript*, McGraw-Hill.

Mernik, M. and Zumer, V. (2003), 'An educational tool for teaching compiler construction', *IEEE Transactions on Education* **46**(1), 61–68.
URL: *https://ieeexplore.ieee.org/document/1183668*

Nielsen, J. (1994, 2020), '10 Usability Heuristics for User Interface Design', `https://www.nngroup.com/articles/ten-usability-heuristics/`. Last accessed: 2021-04-16.

Rosin, R. F. (1977), 'A graphical notation for describing system implementation', *Software: Practice and Experience* **7**(2), 239–250.
URL: *https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380070214*

Ross, R. J. and Grinder, M. T. (2002), Hypertextbooks: Animated, Active Learning, Comprehensive Teaching and Learning Resources for the Web, *in* 'Software Visualization', Springer Berlin Heidelberg, pp. 269–283.
URL: *https://link.springer.com/chapter/10.1007/3-540-45875-1_21*

Sebesta, R. (2007), *Concepts of Programming Languages*, 8th edn, Pearson Addison Wesley.

Slansky, J. and Finkelstein, M. (1968), 'A Formalism for Program Translation', *J. ACM* **15**(2), 165–175.
URL: *https://doi.org/10.1145/321450.321451*

Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O. and Steel, T. (1958), 'The Problem of Programming Communication with Changing Machines: A Proposed Solution - Part 2', *Commun. ACM* **1**(9), 9–16.
URL: *https://doi.org/10.1145/368919.3165711*

ter Horst, G. (2014), 'Tombstone Diagrams for Dia', `https://github.com/Ghostbird/Tombstone-shapes-for-Dia`. Last accessed: 2021-04-06.

Watt, D. and Brown, D. (2000), *Programming Language Processors in Java: Compilers and Interpreters*, Prentice Hall.

Wickerson, J. and Brunet, P. (2020), 'Diagrams for Composing Compilers', `https://johnwickerson.github.io/papers/jdiagrams.pdf`. Last accessed: 2021-03-03.