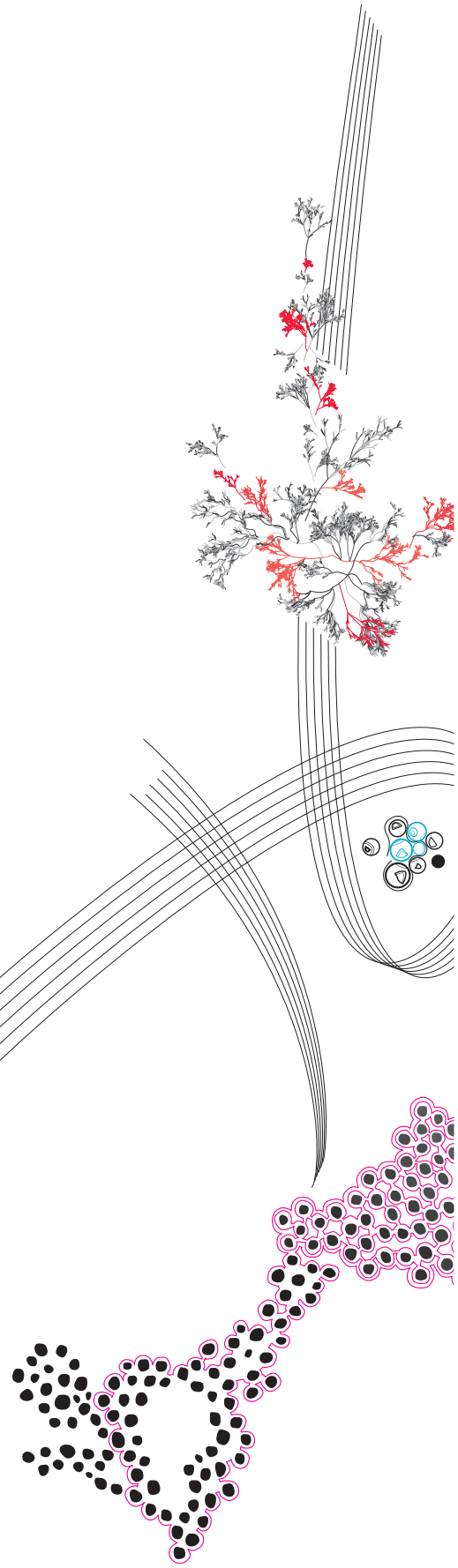


Design Project Technical Computer Science
Group 6

Taking UTML to the next level

Bram Hut, s3195074
Daniel Jonker, s2840529
Hanna Gardebroek, s2800012
Sophie Rijkers, s3143392
Twan Weerdenburg, s2755467

Supervisors: Ernst Moritz Hahn
& David Huistra

April, 2026

University of Twente
Faculty of Electrical Engineering,
Mathematics and Computer Science

UNIVERSITY OF TWENTE.

Abstract

This is a report written for the Design component of the Design Project module. This project has been sent to us from the Formal Methods and Theories department (FMT) of the EEMCS faculty.

UTML (University of Twente Modelling Language) is a web application created by and for UT students that can be used to create various UML diagrams. Students can use the application to make exercises, and teachers can integrate the application within the exam environment of Anubis. The current version of UTML is functional, but far from ideal, and as such we set out to create a better version of UTML from scratch.

During this design project, we identified most problems with the current version of UTML by a form sent out to first year students and asking relevant teachers about their requirements. Using this information, we created a new version of UTML using Vue as a framework and the DGM.js package as a base to draw diagram shapes. While this package had a lot of functionality for drawing, it did not fully support all desired functionalities, and because of this we forked the repository to implement the things we needed ourselves. To make sure this new version of UTML was usable and had all requirements, we did both system tests and user tests.

In the end, we created a version of UTML from the ground up that uses up-to-date dependencies, has an intuitive editor with all functionalities users are used to, and with a relatively easy way to add new diagrams.

Contents

1 Introduction

UTML (University of Twente Modelling Language) is, as the name suggests, a tool to model and create UML diagrams related to courses taught at the University of Twente (UT). These diagrams are often used in information and communication technologies. UTML is designed to support the diagrams that are needed within the UT courses.

This is not the first UTML version. Five years ago, another design project had created UTML, which was later maintained and improved by the UT. However, this version lacked a lot of behaviour that users would expect from an editor, which resulted in students being confused and angry when they had to use the application for exercises in the second Technical Computer Science and Business Information Technology module.

Our task was to improve this version of UTML by both getting it up-to-date with modern libraries and adding new functionalities. After some investigation, we decided not to update the existing version, but to start from scratch and recreate the entire application within a Vue.js framework and using the DGM.js library. This library is designed for creating UML shapes and provided us with a shape editor, and thus a solid base to build our project on. However, the editor did not have all functionalities we desired and because of this we decided to fork the repository in our project so we could easily implement our requirements.

Having a solid base, we started gathering requirements. Some requirements were already gathered from the old UTML and from complaints by teachers and students. During the design process we consulted the module 2 guidebook and slides to make sure all diagrams are complete, and verified these diagrams with module 2 teachers. When the base implementation of UTML was finished, we performed user tests to further check if the behaviour of diagram components is as expected and also to help us with UI design choices.

This report will describe how we created the new UTML and the decisions we had to make along the way. First, we will elaborate on our decision to recreate the application instead of updating the new version, after which we show how we acquired all necessary requirements for both the application design and which diagrams we should have in UTML. We then move on to an explanation of how we performed user tests and what results they gave, and after that we explain how we designed the UI and the code itself and elaborate on certain decisions that we had to make during the project. In the section after that we take a look at system tests to see whether our application is functioning as expected followed by a risk analysis. Finally, we reflect on our process and what could be improved in the future.

2 Redesign

In this section we will briefly talk about our decision to scratch the old UTML and create a new version.

One of the big questions our group had to face before starting the design project was whether or not we were going to start from scratch. This was not originally the plan, but after looking at the code of the old UTML we started to consider the option of remaking it.

After our first meeting with our supervisors we got a clearer idea of what was required of this project. One of our supervisors mainly requested that UTML would be more reliable; that it had less bugs and that it could work well in an exam environment. Our other supervisor wished the project would be easily maintainable and had updated libraries. To update the old UTML would require updating 10 major versions of the Angular framework, a task which would take a long time and after which we would still be left with a project that had many foundational bugs and issues that needed fixing.

Furthermore, since we are not the original creators of the old UTML, some design choices were made that we did not agree with or did not understand. We found the code's structure to be a little odd at times, and although the maintainer did a good job of updating UTML from the previous design project and making it somewhat usable, the bugs were hard to find and correct.

Because of all these points, we decided to try to convince the supervisors of our idea to recreate UTML. To start from scratch with new libraries, a new framework, and a new design. This would, if done correctly, meet all of the set requirements of having a reliable version of UTML, that can be used in exams and that would be easy to maintain.

To achieve our goal of being allowed to remake UTML we tasked ourselves with creating a prototype to showcase our idea in the second week of the project.

This involved rethinking the design to make it more user-friendly, thinking about which frameworks and libraries would be suitable, and talking with teachers and students to find out what they would like to see improved in the old UTML. While part of our team started working on the new prototype, the rest started gathering requirements and planning interviews.

This is, of course, a little bit of a backward way of designing, starting to work on the prototype, and simultaneously gathering requirements. But at this point, we had already explored the old UTML and had read through the problems it had. Moreover, we had already talked to our supervisor to see what his requirements for the new system were, meaning that we had a solid base for starting our prototype.

Some of the choices made at the beginning would ultimately have to be revised or removed, and some things we did not think about thoroughly enough at the start had to be redesigned later in the process. *More on this in sections UI Design and Technical Design.* Even with this unconventional design process, the base of our new UTML was good enough to convince our supervisors of the rework, and in the second week of our project we could officially start to work on UTML2!

3 Requirements

This section will delve into the methods of gathering requirements and their results. These requirements had largely been gathered before making the decision to switch over to the new version of UTML, and served as a basis of what that new version of UTML should contain.

The order of the requirements is arbitrary. Additionally, footnotes have been placed where changes have been made from the original requirements.

3.1 Requirement elicitation

During the requirement elicitation of this project, we have used the following tools to find our requirements:

- A good observation of the current UTML app, expanded by the requirements of each supported type of diagram.
- A list of bugs collected by the teachers of the module Software Systems.
- A self-made survey which was sent to students who used UTML.
- User stories for the different stakeholders
- Conversations with teachers and future maintainers

The results of the survey were analysed and compared with the list of bugs received from teachers. Using this method, we were able to create a complete overview of the current issues with UTML, the parts of the system that people enjoy, and what features are missing in its current state.

The table in *Survey results* showcase our findings. These tables have all issues raised by teachers, and a count of how often we saw those issues in our gathered feedback. We also added some rows for feedback that was not gathered by teachers, and rows for what features students liked, since this was not highlighted in the feedback from teachers.

Moreover, to better understand the wants and needs of the stakeholders, we created multiple user stories. These can be found in Appendix B. Using the user stories together with the analysed survey results we wrote the requirements report.

3.2 Functional Requirements

3.2.1 Appropriateness

A01 The user should be able to create diagrams from scratch using the software. [1]

3.2.2 Completeness

CP01 The maintainer should be able to add new types of diagrams to the software with minimal code changes.

CP02 The user should be able to add, remove, resize, move, and colour objects on the canvas. [2]

[1] supported diagrams can be found in *Diagram requirements*

[2] "colour objects" has been removed due to not being a 'must have' requirement and cluttering the UI too much

CP03 The system should provide alignment and snapping tools.

CP04 The system should allow users to add and edit text labels on objects.

CP05 The user should be able to export their projects as a UTML file, PNG, or SVG.

CP06 The user should be able to import UTML files.

CP07 The system should cache recently created files in the web session.

CP08 The user should be able to access recently cached files.

CP09 The system should support single-diagram environments.

CP10 The system should be able to evaluate finite state automata. [3]

CP11 The system should be able to integrate with examination software, including Anubis.

CP12 The system should support an "exam-mode" during exams that is limited in functionality according to the specification of the teachers.

3.2.3 Correctness

CR01 The system should handle user input as expected from similar graphical programs.

CR02 The system should not add, displace, or remove objects without the intention of the user.

CR03 The system should correctly evaluate automaton diagrams. [3]

3.3 Non-functional Requirements

3.3.1 Performance

PR01 The system should not take more than 3 seconds to load the interface on standard broadband.

PR02 The system should have no more than 100 ms of delay between the action of the user and the feedback of the desired result.

PR03 The system should support diagrams with at least 500 objects without noticeable lag on at least current midrange processors. [4]

3.3.2 Compatibility

CM01 The system should function correctly on Windows, macOS and Linux.

CM02 The system should work on all recent university provided laptops and chromebooks used in exam environments.

CM03 The system should work on all the browsers supporting ES2024 and higher.

CM04 The system should be compatible with the Anubis examination program.

[3] This was deemed out of scope for the project

[4] DGM.js was a bottleneck for this as discussed in Improving internal DGM.js update pipeline

3.3.3 Usability

US01 The system should be intuitive to use and usable without training.

US02 The system should accept the most common keyboard shortcuts.

US03 The user should be able to zoom using the trackpad, zoom buttons, and mouse.

US04 The user should be able to move around using the trackpad.

US05 The system should provide non-intrusive tooltips and an onboarding tutorial when starting up the interface. [5]

3.3.4 Reliability

RE01 The system should not crash.

RE02 The system should cache recent diagrams in case the page gets reloaded.

3.3.5 Security

SE01 The user should not be able to leave or enter "exam-mode" without permission of a teacher.

3.3.6 Maintainability

MA01 The system should be modular and follow best practices.

MA02 The system should support automated testing for core features.

MA03 The system should allow new diagram types and shape libraries to be added with minimal code changes.

MA04 The system should have ESLint enabled.

MA05 The system should have a maintainer's manual.

3.3.7 Portability

PO01 The system should support web-based access without installation

PO02 The system should support import options for previous versions of UTML. [6]

[5] We ended up making a help-menu instead, as it is less intrusive

[6] This was, in discussion with relevant teachers, deemed out of scope for the project

4 Diagram requirements

Since the most important function of UTML is that users can create diagrams with the software, we have spent a long time figuring out what diagrams should be supported and what features they should have.

Priority was put on the diagrams that are used in Module 2 and Module 7 of the bachelor Technical Computer Science: Activity diagrams, Class diagrams, Use Case diagrams, Sequence diagrams and Automata. These were created first, before moving on to the other diagrams. In the original UTML there were more diagrams, but in conversations with the teachers who would use these diagrams, some turned out to be unused and thus were left out of our rewrite. To find all diagrams and our specifications, see Appendix C.

In this chapter we will look at how we gathered the diagram requirements and our main changes from the older versions of UTML.

4.1 Gathering diagram requirements

At first, we figured that gathering diagram requirements would be easy, since we thought we could simply copy what was already in UTML. However, as it turned out, the old UTML was missing a lot of objects that should have been added to certain diagram types (e.g. aggregation or composition arrows for a class diagram, or a lost/found message in a sequence diagram). Also, it had some objects that should not have been implemented at all (such as a clock item in the use case diagram).

When we realised we could not trust the old UTML to have all the correct objects, we decided to schedule interviews with teachers and TAs to figure out what objects were missing or incorrect.

We interviewed the main teacher of the design component of Module 2: Software Systems. She gave us a long list of things that were currently missing from diagrams. She also suggested we look at her slides and the module guide from module 2 to check if we could recreate every diagram from the manual. These slides and module guide were a great source of information for us. She also mentioned some quality of life things that she would like to be improved, such as the way "alt" sections in a sequence diagram are calculated.

We proceeded with talking to the teacher of the Languages and Machines course, who also happened to be one of our supervisors. Our goal was to gather some more requirements for the Automata and Fault trees. Finally, we interviewed a teacher from Electrical Engineering, to see if it is possible to include diagrams for his course in the future. We decided that while this is definitely possible, adding these diagrams would be a low priority for us, partially because the teacher already has his own tools for creating Electrical Engineering related diagrams.

4.2 Technical considerations

We planned to completely rewrite UTML, using the DGM.js library. Certain diagrams require the implementation of specific interaction behaviour, which had technical implications. This meant that these behaviours required changes to the codebase to enable them. In this section we will discuss what technical requirements are needed for this process.

4.2.1 Class diagram

- **Configurable components**
  Within the old UTML, the user has the possibility to change a connector based on

its type. It would function as a drop-down with the connector type. In addition, some components, like the class component, need to be interactable in the editor. The user should be able to change component properties, like e.g. class type, or class methods. For this, some system should be created to handle this behaviour. The technical design of this can be read in section Computed Components.

4.2.2 Activity diagram

- **Vector data**
  The activity diagram has a few shapes that are created by an SVG path, since it cannot be created by simple shapes. It took some research on how to implement this, since it turns out that DGM.js has four different ways of rendering scalable vector data.

- **Connectable options**
  For certain shapes, we want to allow connections to only some parts of the shape, so we had to create a custom shape property, called `connectableOptions`. It defines which points an arrow can connect to, like `corners-only`, `midpoints`, `center` or `outline`.

- **Configurable components**
  See Class diagram

4.2.3 Sequence diagram

Another diagram type that required a lot of "special behaviour" was the sequence diagram. Users often complained that, aside from missing many needed objects, the old sequence diagram was not intuitive to use and had a lot of unwanted behaviour. For example, when adding an execution block to a lifeline, the block would have to connect to the lifeline and a new lifeline would have to be added under the execution block. Not only was this a tedious process, it also meant it was hard to get lifelines fully vertical. Additionally, it was only possible to snap connectors to predefined points on the execution block, limiting the usability of the tool to create proper sequence diagrams.

- **Lifeline**
  In the new UTML we implemented a different approach: the lifeline object consists of one large shape composed by an image of the lifeline type (actor, object, etc.), the name of the lifeline, and the lifeline itself. The lifeline is one dotted line that cannot be connected to and can only be resized vertically. Execution blocks are also only able to resize vertically and can be added to the lifeline, and connectors can be connected to the execution blocks. These connectors can connect to any point on the execution block.

  This results in behaviour where execution blocks are always nicely aligned in the lifeline and move together with the rest of the lifeline. Only when an execution block is added to the lifeline it is possible to have a connector connect to it, and the lifeline type can be changed from the property panel.

- **Fragments**
  Sequence diagrams also require fragments. In the `Alt` fragment user must be able to add multiple fragments to indicate a logical split based on a condition, where it must be easily accessible. In the old UTML, users are required to enter a pixel-offset

to indicate the positioning for each fragment. In our version, we have improved this by creating a handler, with which the user is able to drag the edges of the fragments visually. In addition, we added an anchored text to each fragment. With this anchored text, the user is able to control the positioning of the fragment condition visually and easily.

- **Connectable options**
  See Activity diagram

4.2.4 Use case diagram

- **Connector labels**
  Use case diagrams have a unique property with their connectors, which are the `include` and `extend` connectors. These connectors are not differentiated by their arrow shapes as is the case with other connectors, but by their labels. This means that, when one of these connectors is selected, the middle label should always be `«extend»` or `«include»`. Since our UTML follows that the connectors should also be changeable to the other connector types that do not have an inherent label, the label should disappear from the `include` and `exclude` when changed to a different connector. Additionally, when it is the other way around, the connectors should overwrite the middle label to say either `«extend»` or `«include»`.

- **Configurable components**
  See Class diagram

4.2.5 Automata

- **Arc shape**
  For automata the most noteworthy behaviour is the arc shape and snapping to the state symbol. Snapping is easily adjusted and implemented because there is one simple piece of code responsible for this. It also had to be adjusted for the other shapes, so this is no issue. The arc shape is a bigger technical problem. The DGM library already supports arrows that use a Bézier curve. Unfortunately, it turned out this was unusable due to the heavy computations it requires and the unoptimized update pipeline the library has. The behaviour was also not exactly as expected. The middle point of the curve did not move when an endpoint was moved. To fix this, we either had to rewrite the update pipeline which is a major task. But then we would still have to make the user experience better.

  In the end, we chose to make a new kind of arrow that uses a more efficient parabola formula for its path. This arrow also has a control point in the middle to change the curve that also moves when an endpoint is moved. By doing this the performance was much better. It is also less tedious to move an arc now, because you don't have to move the middle point each time.

4.2.6 State machine diagram

- **Vector data**
  The state machine has a single component with an SVG path. See Activity diagram

4.2.7 Computer architecture diagram

The computer architecture diagram did not require any technical implications, since it has a single component, which is a simple rectangle shape.

4.2.8 Fault tree

- **Vector data**
  Fault Trees have simple shapes. It requires SVG support for the gates to be added, luckily this is no problem. Furthermore, the arrows are already supported by DGM.js. See Activity diagram

- **Snap points**
  Fault trees have SVG shapes on which connectors need to snap to varying positions. For this, a SnapPoint shape needs to be added to support a custom snapping position, which can be configured per-shape

- **Infinitely many configurable snapping points**
  For fault trees, there should be infinitely many available snapping points on a shape, but these points should all be easy to parse and visally intuitive. As mentioned in the reflection, namely New diagrams, this last requirement is not implemented.

5 User Testing

In this chapter we will show our test plan and the results and take aways of that testing.

5.1 Test Plan

Since one of the limitations of the previous version of UTML was the ease of use, we focused a lot on user testing during this project. Before testing started, we made a test plan to send to the Ethics board of the University of Twente, to give us permission to start testing. We also made an informative document that serves as a consent form for testers to sign; this form can be found in Appendix D.

After getting permission from the Ethics board we started user testing. This was done in a structured manner. We tried to test with two people from the group present, one to take notes and the other to ask questions, or just one person if no-one else was available. We started by letting the participant recreate a diagram from the module 2 design manual. These diagrams were randomly selected and either a Use Case Diagram, Class Diagram or Sequence Diagram. The diagrams asked to recreate are shown in figure 1.

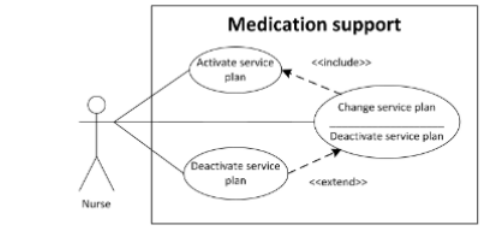


Figure 2.2: Partial Use Case Diagram for medication support (Exercise D-2.4)

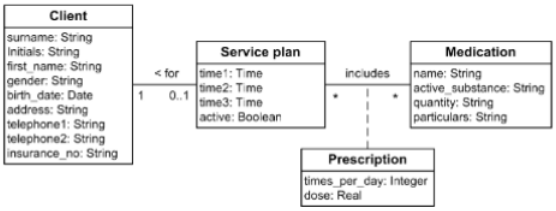


Figure 2.3: Partial Class Diagram for medication support (Exercise D-2.4)

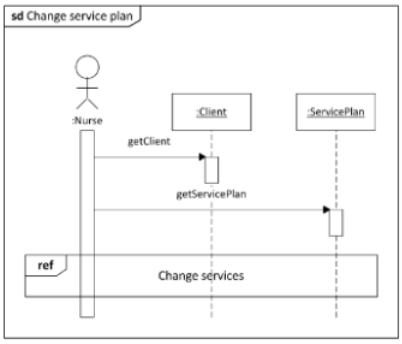


Figure 2.4: Sequence Diagram for *Change serviceplan* (without control object) (Exercise D-2.4)

FIGURE 1: Diagrams from Module 2: Software Systems manual

After this, the participants were asked some standard questions to see how intuitive our design was, such as "can you export this file as a .png?", "Can you make this text bold?", and "Can you change this connector type?" We wrote down how long it took them to do these things and if they made any mistakes while trying to accomplish the given tasks.

Finally, we asked them some more general questions about the design. Such as if they enjoyed the colours, if they found the movement and zooming easy to use and if they found the shortcuts intuitive. If the participant had used the old version of UTML we asked them to compare the two versions and to tell us if they thought anything from the old version was missing.

We wrote down all of our findings and later compiled them into one big document listing all the things we found were worth discussing. This testing and design process was iterative, if we found any major but easy to fix issues during a test we discussed them with the team and fixed them before the next user test. This way, we ensured that we did not

end up with everybody mentioning the same issues. The testing process also went through multiple stages, lasting a few weeks.

5.2 Student test results

We started by only letting students test our system, these students were volunteers who were either first-year TCS/BIT students, Teaching assistants (TAs) or students with a background in website design. We interviewed 11 students over the span of two weeks. There was some time between interviews so we could fix bugs and issues that came up.

There were a lot of small feedback points and bug reports, such as "The search icon in the components tab is off centre", "Some shortcuts are under the wrong category in the shortcut tab" and "Actor disappears when switching to dark mode" (the image had a black outline and did not change when switching to a black background, making it difficult to see). The main issues that arose during testing were as follows:

- **Class diagrams are difficult to fill in, students can not find where to edit the name of the class. They often double click on the class and are confused as to why nothing happens.**
  After discussing this issue we found that it was caused by two factors: first of all, during the initial tests our properties panel was on the right side of the screen, behind the components tab. This made it difficult to find. Second of all, users expect to be able to double click on text to edit it, this was not yet implemented when testing.

- **The attributes and methods menu gets very large**
  When testers were asked to recreate the class diagram as shown in figure 1. They often ran into problems with the size of the attributes and methods menu. To fix this, we made the fields smaller and collapsible.

- **Interface and Abstract types missing from class objects**
  This was a note from one of the Teaching assistants who noticed that two types of Class diagram objects were missing. We added those objects after that.

- **Users can not find the actor object for a sequence diagram**
  We decided early on in our design process to do sequence diagrams differently than the old UTML did them. In the old UTML, lifelines, actors and the dotted line belonging to lifelines were all separate objects. This was met with a lot of feedback from students who found that creating sequence diagrams took a long time because all components needed to be added separately (and often did not align properly). We therefore made them one object, but students who were used to the old UTML found this confusing. We decided that this issue was not very important, as the system will be used by new students next year who are not used to the old UTML and who would probably not find this confusing.

- **It would be nice if you could click on an object and the option to add an arrow starting from that object would appear**
  This feature is often contained in other diagram editing tools. Because of the way we implemented arrows, adding such a system is difficult. It would be a big quality of life update so we understand the wish, but after talking the idea through we decided focussing on more prominent issues, instead of adding an additional way to create arrows.

- **Dragging and dropping items into the editor is not very intuitive**
  At the start of testing, when asking a user to add their first object, we noticed many people double clicked on an object instead of dragging and dropping. This was not behaviour we had thought much about and it was surprising to see that it was not found intuitive. We decided to add an option where double clicking on an object makes it appear in the middle of the editor. As a side-effect, this also accidentally improved touch screen support, because tablet users could now also add items to the editor. This was, however, not a priority for us and we did not attempt to improve this support further.

- **The properties panel is difficult to locate and switching between components and properties is tedious**
  This was by far the most commonly received feedback. We needed to revise the way the properties panel worked, currently it was located behind the components tab on the right-hand panel. Users had to manually click on an object, then on the properties panel and then edit the desired property. To come up with a better solution, we decided to start AB testing. This is discussed in the section "A/B testing" below.

5.3 A/B testing

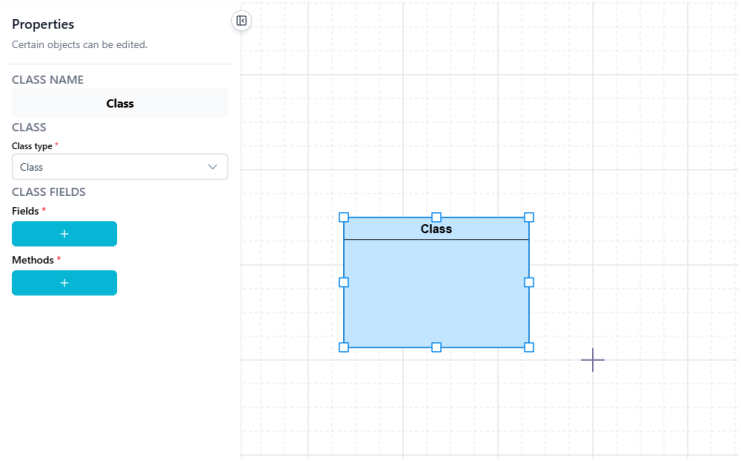
After concluding that we had to change the properties panel, we consulted with our supervisors and decided that we wanted to try A/B testing. We created 2 different versions of the properties panel, one with a sidebar on the left-hand side of the screen and the other with a pop-over properties panel. Both Properties panels would only open when the user selected an object that had editable properties, and close when the user clicked out of them. The designs for these different panels can be found in figure 2.

We did some more user tests, this time the tests only focused on the properties panel. We also asked some people who had already tested the previous iteration to test again to see if they thought the properties panel had improved. In table 1 are the main findings of these tests.

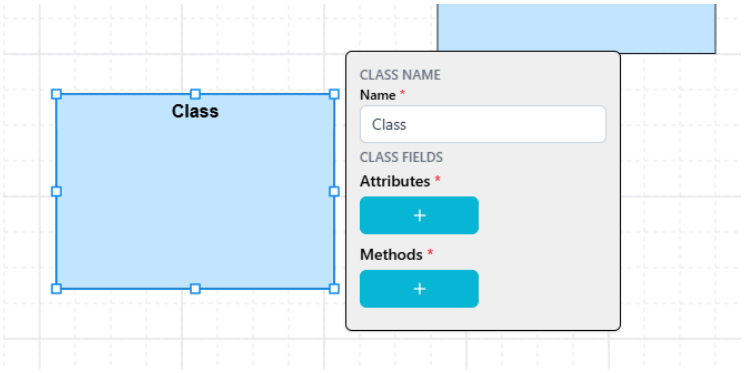
Feedback	**Version A**	**Version B**
Placement of the panel	Predictable, sometimes blocking part of the editor	Unpredictable, sometimes blocking part of the editor
Size of the panel	Large, often too large when only editing one item; however a good size when editing many methods and attributes	Nice size, but too small when editing many methods and attributes
Appear and disappear behaviour	Appears and disappears on click; can also be manually opened and closed	Appears and disappears on click, but cannot be manually opened and closed

Table 1: Comparison of Version A and Version B

Clearly, both versions had good things going for them. But ultimately, we had to make



(A) Properties in left-hand sidebar



(B) Properties in pop-over

FIGURE 2: Version A and Version B

a decision. We decided that although both versions of the properties panel took up a lot of space on the editor when opening, the placement of the pop-up panel made it less suitable for the project. Users appeared annoyed when the pop-up blocked part of their editor and less so when the left-hand panel opened. This was because the panel opening was more predictable. For us, this design was also more beneficial. Having a fixed size, makes it easier to design the side-bar.

5.4 Teacher testing

Finally, after having completed the user testing with students and fixing the main issues that arose in this process, we sent UTML2 to teachers for testing. We wanted to see what they thought about the product since they would be using it in their courses. We waited with sending it to them until we had a MVP since we did not want to waste their time having to write down bugs we were already aware of.

Once again, the head teacher of Module 2 Design was a good help in this. Her feedback was mostly positive, with a few minor points that we changed. Such as a little whitespace between methods and attributes in a class diagram, and renaming the "connector" to "transition" in the state machine diagram. She also had a lot of feedback on objects that were in UTML2 that she did not need for her course. She suggested we remove them. However, we decided that we would rather have a complete diagram making tool than only have the objects used in module 2. This is mostly because we think that the product is

more future-proof this way. If the teachers of Module 2 ever decide to make a different manual with different diagrams, they will now have more options to choose from. Having a more complete diagram tool is also useful for modules besides module 2. During their studies, a student will have to do a lot of projects that involve creating diagrams. By having more options UTML is now a more attractive tool to use for the creation of those diagrams. Lastly, The teacher had some ideas that we sadly did not have time to implement. These can be found in the Future Works section of this report.

We also contacted three more teachers involved with module 2 design to ask for their feedback. They all seemed positive, with only a point being raised about Remindo integration, an exam environment used by the University of Twente. So far, we had thought about Anubis integration, a different exam environment, but not about Remindo. This was because our supervisor wanted to work with Anubis. We contacted the current maintainer of UTML and future maintainer of UTML2 who told us that he did not think this would be an issue.

Lastly, of course, we kept testing with our supervisor and future maintainer. These tests were not structured but more on an "if you find something, let us know" basis. We kept in contact with them using Discord, where they could always message us if they found anything. This turned out to be a very helpful process, and we found a lot of bugs and issues this way.

6 UI Design

During the development of an interactive web-app with many user interactions, design becomes an important problem. A part of the design decisions made have already been specified in the user testing section, but in this section of the report we want to highlight more aspects of the general design.

6.1 Usability

We had a few important points in mind when designing this system. First, it should be intuitive for the user. Second, for the user, the use should not differ too much from its predecessor (and what is changed should be seen as better by most users). We came to these points because the end-users of our application will be varied; students can have different experience levels in making UML diagrams, and teachers can be from different fields. Therefore, the application should be available and easy to use to a wide range of people. We figured teachers might also be more willing to use the new application if it behaves similarly to the old app, because there is less for them to adjust to.

For the next part of this section we will highlight different parts of the product and showcase their design. The full web-app is shown in figure 3.

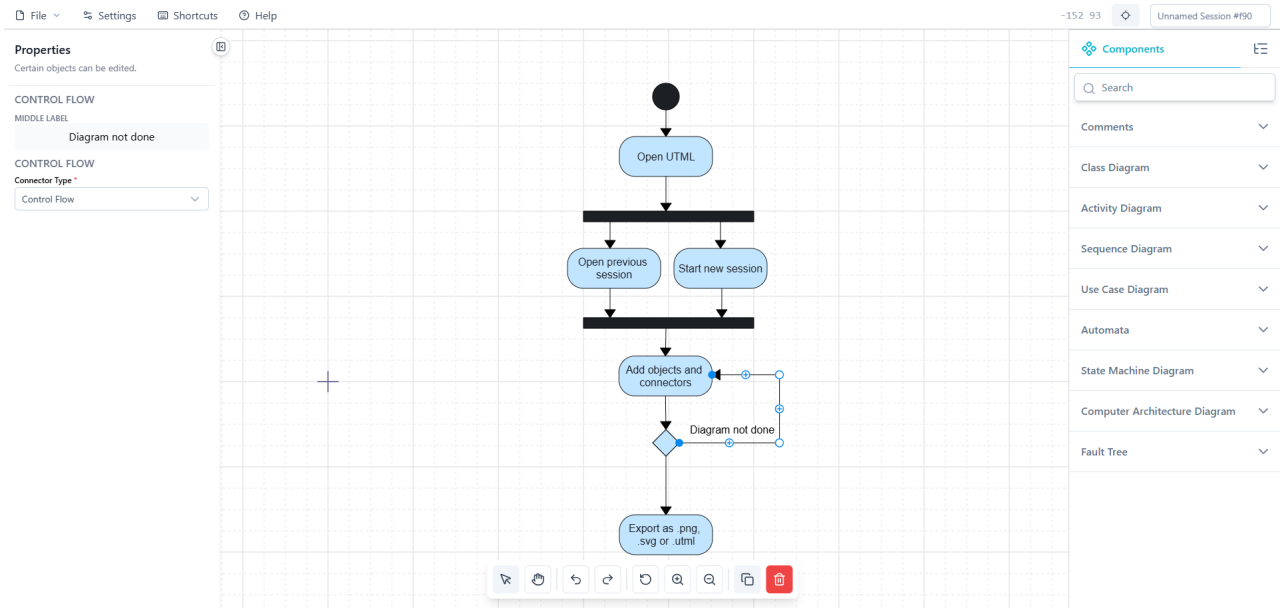
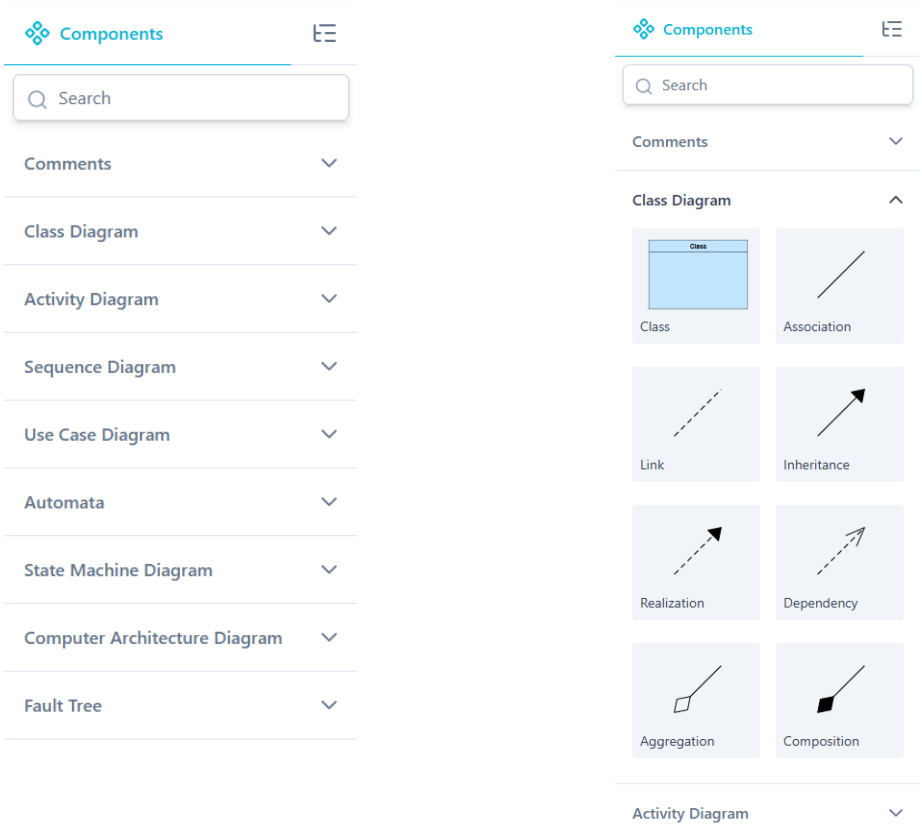


FIGURE 3: Editor with a diagram for reference

6.2 Components

When opening the editor, the user will find a components tab to the right hand side of the screen, see figure 5. This side bar shows the different types of diagrams that can be created using UTML2. Once the user has decided which diagram they want to make, they can click on the fold out button to see all objects belonging to the diagram they wish to create. This design was copied from the old version of UTML with a few small tweaks:

- We added a search bar so a user can easily find an object they are looking for



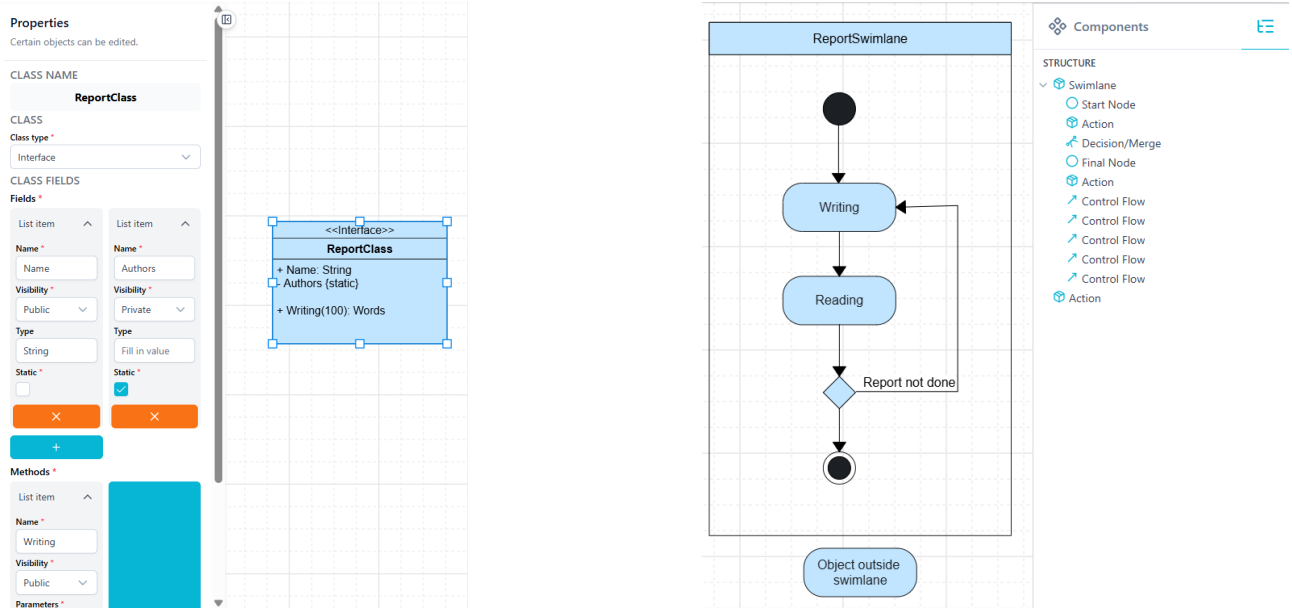
(A) Components panel (B) Class diagram folded out

FIGURE 4: Components panel

- Drop down symbols were added so the user can more easily see that the diagram types unfold
- Instead of only showing one object per line (as the old UTML used to), we show two objects per line to make the components bar take up less space. This way the user does not have to scroll for a long time to find the object they need.
- We chose to show all possible types of connectors instead of only the basic one. In the old UTML, when the user wanted to make a class diagram for example, they had to select the common arrow connector, and then change it to a different arrow using the properties tab. The new UTML shows the user all options for different connectors so they can more easily see all their options.

6.3 Properties tab

The properties tab, as seen in Figure 5a, opens up on the left hand side of the screen when a user selects an object that has editable properties. These objects can for example be connectors, which can have different arrowheads and line types. Another case is the class diagram where the properties tab serves as the place where a user can add methods and attributes to a class. The properties menu has seen many different iterations throughout the making of UTML2. As has been mentioned before in the A/B testing section of this report.



(A) Properties panel (B) Filetree

FIGURE 5: Property panel & Filetree

We also went through multiple different ideas of what properties should be editable. At first the properties tab also had options to change the colour of objects, a field to change the size in pixels and a field to change the text in objects. During user testing, we discovered that users did not feel the need to ever change the colour of an object or resize it using the properties tab. Thus we decided to get rid of this functionality. Users also expressed a desire to have inline text editing instead of having to edit using the properties tab. This was implemented and we think it has improved UTML2 greatly.

6.4 Filetree

One of our goals was to communicate to the user the component structure, since we were dealing with containable components, like a swimlane. Since our structure is recursive, parent-child, a tree was the best way to display these relations, as seen in figure 5b. We opted to design a filetree which was similar to other programs. To serve additional purpose, if a user clicks on a component, then their editor gets centered on that component and the component gets selected.

During user testing, we found that there are not many people who find this feature useful, so we have decided to hide the filetree behind the components tab. This way, teachers or other people interested in seeing which components are in a swimlane can still see it, but it is not cluttering valuable editor space.

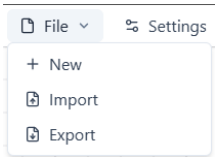
6.5 Top menu

At the top of the editor is a small grey menu with four buttons, as can be seen in figure 6a. The buttons functionalities are as follows:

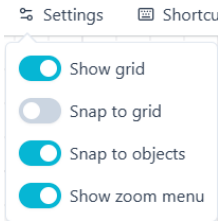
- File: here a user can create a new file, import an old .utml file or export there current file as a .utml, .png or .svg file. See figure 6b.

- Settings: a settings menu where users can (dis)able seeing the grid, objects snapping to grid, objects snapping to objects, and the zoom menu. See figure 6c.
- Shortcuts: here the users can find an overview of most of the shortcuts implemented, see figure 6d.
- Help: behind this button the help menu can be found. It has six subsections titled: Basics, Movement, Items, Connect & snap, Saving and Contact. See Figure 6e. Behind every button a short manual can be found on the corresponding subject. After pressing the contact button users can find the contact information of the future maintainer of UTML.

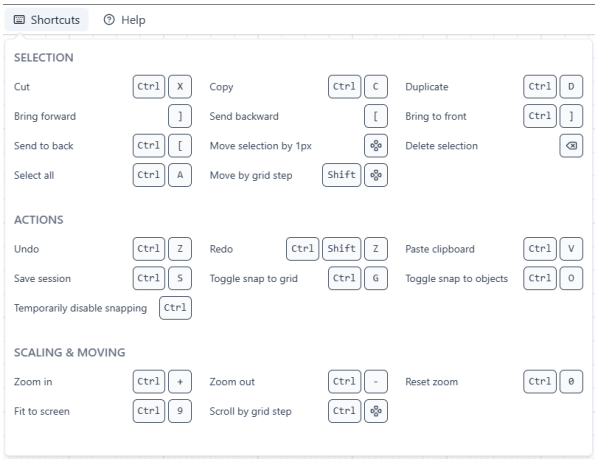
(A) Top menu



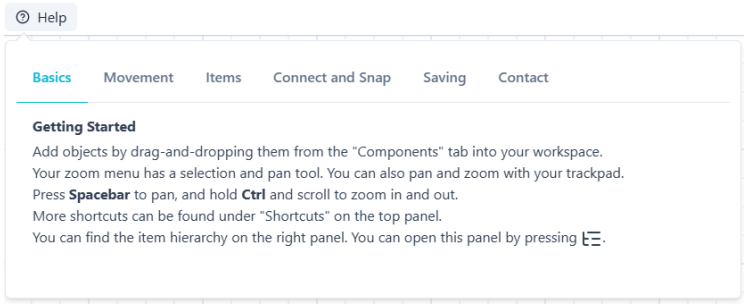
(B) File button



(C) Settings button



(D) Shortcuts menu



(E) Help menu

FIGURE 6: Top menu

6.6 Bottom menu

At the bottom of the screen users can see a floating menu with a few buttons on it. See figure 7. The buttons functionalities are as follows:

- Select: when a user is in select mode, they can click on objects and connectors and edit or move them. They can add components via the components panel and edit their properties in the properties panel.
- Move: when a user is in move mode, they cannot edit objects and connectors and they can not add new components. They can click and hold anywhere on the canvas and move their mouse to move around. This functionality is not needed when using a trackpad, but is implemented so that mouse users can move from side to side more easily.
- Undo and Redo: these buttons either undo your previous action (adding, moving, deleting or changing the properties of an object), or redo a previously undone action.
- Zoom in, Zoom out and Reset zoom. These buttons either zoom in or out on the canvas, or can be used to reset the zoom back to the starting state of the canvas (100%). In previous iterations there was also an indicator for percentages, but this was found to be confusing and unnecessary by testers since the zooming percentages did not update when users used a trackpad or mouse wheel to zoom in.
- Duplicate and Remove, these two options are faded when a user has not made a selection on the canvas. When a user has one or multiple objects selected they can use these buttons to either copy and paste them, or delete their selection.

FIGURE 7: Bottom menu

6.7 Text editing

Lastly, we created a small pop-over menu so users can easily edit text. Because text-editing works from both the properties panel and in-line, the user does not need to use this pop-over. However, we found during testing that users expect to be able to double click on text to edit it. The pop-over menu for text-editing has seen many iterations, but we settled on the design shown in figure 8. The features of this menu are self-explanatory so we will not go into depth in this report.

FIGURE 8: Text editing pop-over

7 Technical Design

This section will talk about the technical considerations that were made during the development of the next version of UTML.

7.1 Libraries

The original UTML stack used Angular and Spring Boot. Barely any libraries were used to simplify the programming process. Due to this a lot of code is manually handled and contains some bugs.

Given our team's experience and also preference of the future maintainer, we chose to switch from Angular to Vue. Vue has a better learning curve and we prefer the file structure over Angular.

In addition, we chose to use PrimeVue with Tailwind CSS for our component library. This makes it easy to add modern pre-made components such as buttons, tabs, and tooltips.

Lastly, at the core of our application is the DGMjs library. It adds a HTMLCanvas-based editor with an infinite canvas. In the previous UTML, all interactions with the editor were manually handles, which caused a lot of bugs. DGMjs now handles all these interactions for us, and also keeps track of the state of the editor.

Internally, DGMjs uses Zod and TipTap. Zod is a statically typed schema definition library, that is used to parse data. TipTap is used for the inline text editing within the editor, because it provides an extensive modular text editor.

DGM.js is licensed under the GPL3 license, this could mean that the source code for UTML had to be open if we were going to use it. This could be a problem, because our client did not want to open source UTML. After checking with the faculty, we found out that there were some clauses regarding distribution that specified that we did not have to release the source code in our case.

7.2 The Editor - DGM.js

The most important library that is used within UTML is DGM.js. It implements a HTMLCanvas-based editor environment where the users can create and interact with (predefined) shapes on an infinite canvas. DGM.js takes an interesting approach to this by providing 'smart' shapes. These smart shapes can be a prototype, have children, constraints and scripts. These are powerful properties that enables the developers to create many different shapes, that consist out of multiple shapes.

The prototype property is used to tell DGM that that shape is a template of which multiple instances can be made. For instance, all shapes that show up in the component sidebar are a prototype, with its children not being prototype. This way you can create bigger shapes from multiple basic shapes as shown in All available Shapes within DGM.js including newly added ones.

Each shape can have multiple children and a parent. This parent-child structure is in place for many components within DGM.js. An open editor has the following structure: it begins with a document where its children are pages within this document. A page's children are all the shapes that are shown on it. Then the children of these shapes are part of a prototype shape or contained within another shape. This structure goes on.

The constraint property allows you to set constraints to the given shape, it is basically a set of rules the shape must keep to. Whenever the canvas changes, e.g. a shape is moved, all of the constraints are checked. If the constraints don't comply, the shape will be changed such that it does. A good example for a constraint is 'align-to-parent'. In this

constraint you can tell a child how it should align to its parent. For instance, horizontally and vertically in the middle. Whenever you move the parent, the child is also moved such that it meets this constraint.

The last important smart property is script. DGM has a custom scripting language `DGMScript` that has support for some basic operations. With this script you can interact with the shapes in the editor and implement custom behaviour you are unable to get with the constraints. To illustrate, you are able to give all children with a given tag a blue fill colour.

We chose this library, because these smart shapes allow us to easily add new diagram symbols. In figure 9 you can see all shapes which can be used to make the diagram symbols. Originally, the project of DGM.js consists out of multiple modules, of which we actively use: core, and export. Core contains the main code for the editor. Export contains a code extension to add support for a few filetypes. The project also implements a simple webpage in React with wrappers for the core in React. We have taken a look at this and rewritten it in Vue. By rewriting it, we can easily integrate it in our Vue environment.

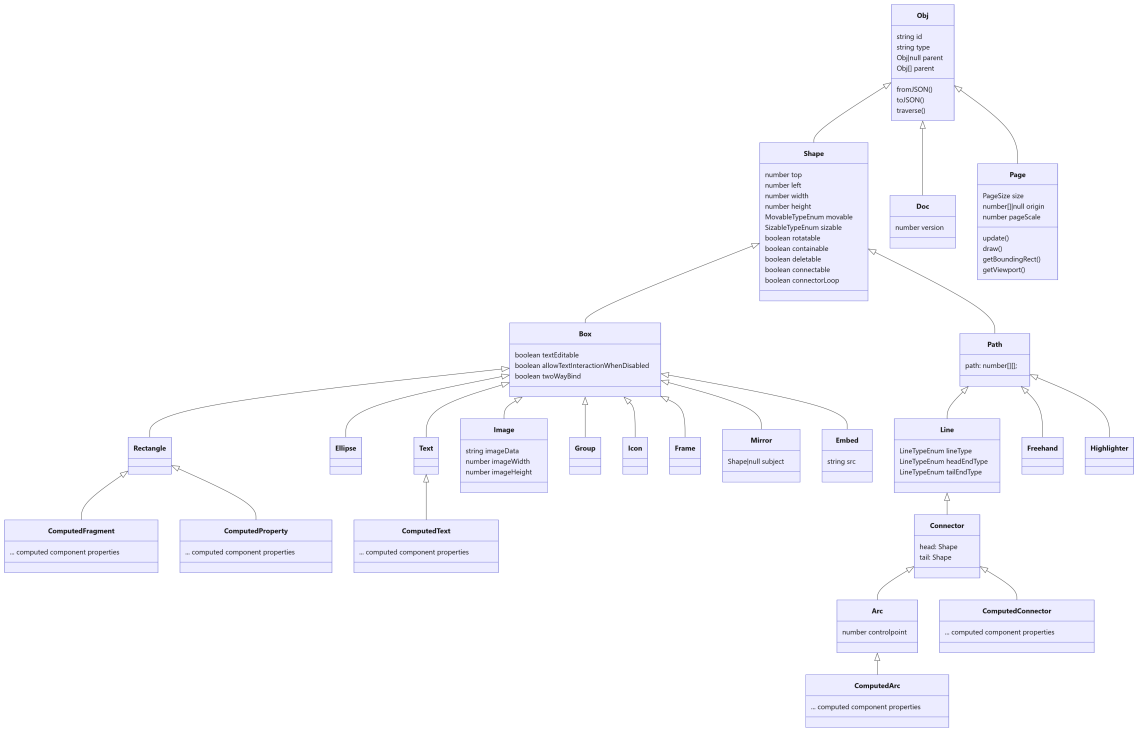


FIGURE 9: All available Shapes within DGM.js including newly added ones

7.3 Component Properties

To support extra features, some classes within DGM.js had to be extended with extra properties. These are used to determine certain type of behaviour or to store data. The following was added:

- **connectorLoop**
  By default in DGM.js, it is not allowed for connectors to have the head and tail connect to the same object, since this would create an infinitely small self-loop. However, we needed this behaviour for the class diagram and the automata, which do self-loops. To achieve this, we had to create a property called `connectorLoop`,

which indicates that a line is allowed to connect to the same object on head and tail. To ensure that connectors do not create an infinitely small self-loop, code was added to modify the connector path.

- **ConnectableOptions**
  By default in DGM.js, you can connect a connector anywhere on a shape. We wanted to make this snapping more logical. However, we needed to change this to reflect how different shapes behave. For instance, it is not logical to show a corner-connection hint for an ellipse. We added the `connectableOptions` property, which is a list of strings. Each string indicates some way to connect to the shape, for instance there is *bounding-corners*, *bounding-midpoints* for rectangle midpoints, *middle* and *outline* which lets the user snap to the outline of a shape.

- **twoWayBind**
  Explained in Two-way binding

- **allowTextInteractionWhenDisabled**
  Many components are built out of many shapes. Because of this, it is sometimes necessary that a shape is visible, but cannot be interacted with, since it is anchored to a parent shape. In that case, the shape `enabled` property is set to `false`. However, if a shape is disabled, containing text cannot be interacted with. For this case, we have created a property called `allowTextInteractionWhenDisabled` and patched DGM.js logic to allow text interaction if this property is set to `true`.

- **componentId**
  Explained in Computed Components.

- **parsedData**
  Explained in Computed Components.

- **deletable**
  By default all shapes are deletable. To disable deletion, you can set this to false.

7.4 Computed Components

One of our requirements was that there should be some components that should change state depending on user input. For example, a connector should be able to change its arrow type when the user changes a dropdown. In addition, this behaviour should be easily adjustable by a maintainer. Because the behaviour should be easily adjustable, we thought that the best way to dynamically define user input was to use a schema. For that, we used the `zod` package, which is a type-first schema definition library with static type inference. It is ideal for TypeScript, since it is typed and it exports many helper functions. These helper functions could then be used to generate a form for a zod schema dynamically. This was done by the sidebar of the editor, which we call the *Component Property Panel*. This property panel would then call some method with new data as soon as it is updated.

Since we knew that we needed to go from user input to some state, we needed to define a function that could take in user input (according to a zod schema) and then return some state, which could be reflected in the editor. Internally, we call this a *compute method*, because it computes some state according to a filled-in schema. Since these states (like which 'type' of arrow is needed, or which label a connector has) need to be reflected by a Shape in the canvas, we decided that the best approach is to create new Shape types

that could handle the state returned by the compute method. To do this, we had to create new Shape classes, which would extend another Shape. For example, we created a `ComputedConnector` class, that extends the `Connector` class to keep its behaviour, but also to add support for the schema.

These custom classes all share some behaviour (to process a schema and reflect it in own state, thus reflecting it in the canvas), but all these classes extend a different Shape type class. This meant that we had to create a Mixin[7], where we input the to be extended class type as a variable, ensuring that a class is created with different base classes, but maintains specific behaviour. We ensured that all the to-be-extended classes needed to extend the `Shape` class, to make sure that we could utilise as much DGM.js behaviour as possible.

In the mixin class, we expose a method called `parseAndSetProperties`, which would be called as soon as the user would fill in the schema. In essence, `parseAndSetProperties` would take the filled-in schema, validate it, parse it to a state and then store it inside the class. Using this, our computed classes could then use this state to adjust their `render()` function, which would reflect the changes back to the user. To define these schemas, we created a global registry to define schemas and compute methods simultaneously. To know which specific component instance connects to which schema, a `componentId` property was added.

The only missing part is how the user interacts with the zod schema. We created a Vue component, called `ObjectFieldComponent`. When a shape is selected, it's children are recursively searched for schemas. Then, for each found shape containing a schema, it is then rendered to the sidebar. Depending on the schema specification of the shape, different things are rendered. Firstly, the zod schema shape is extracted. The type of the schema and the schema settings are stored in a so-called `schema shape`. The schema shape defines some properties, like if the schema is read-only, default values, min/max values, if the schema is required and the schema type. If the schema type is a primitive (a non-object, like a string, number, enumeration or date), then it is rendered as a PrimeVue input component. However, if the schema is an object, then the sidebar loops over it's keys and renders the keys recursively. If the schema is a list, then the sidebar renders a list wrapper around the list values, and those values are then again rendered recursively. As soon as values are changed, the input data is validated according to the schema (zod has extensive support for validation) and the data is then given back to the mixin class by using `parseAndSetProperties`.

In addition, all parsed data is stored as a shape property as `parsedData`, such that all data is persistent accross saves.

The schema shape is very easy to define and it is highly customisable for any maintainer. By only writing a schema and a method, any maintainer can create a component that can change state based on an easily definable schema.

7.5 Two-way binding

We wanted to implement two-way binding, which is a way to edit text, either in-place or in the sidebar. This was necessary, since some text could be edited in-place, like for connector labels, but some text could only be edited in the sidebar. To make the editor more intuitive, text should be editable in all expected places. To achieve two-way binding, we decided that there should only be one single source of truth, namely the `text` property of a Shape, which is where DGM.js stores text in a rich-text JSON format. This format is

[7]More can be read at https://www.typescriptlang.org/docs/handbook/mixins.html

defined by a library called *TipTap*. DGM.js makes use of it for it's rich-text capabilities, namely the in-place text editor, which users interact with by double-clicking any text in the canvas. This in-place text editor also exposed a menu with formatting options (like bold, colours, font-size, etc). So, all we had to do is to extend the Computed Components panel to also include some forms for text. We adjusted our search to not only look for schemas, but also to look for objects that contain a text field. If the object was disabled or not visible, it would be skipped. Additionally, there were some objects which were disabled, but we wanted them to be editable. For that, a component property called `twoWayBind` was created. It overrides the check for disabled objects.

For all objects that contain this text field and where conditions were met, a TipTap text editor is created in the sidebar. In addition, the text formatting toolbar is vertically placed above the text editor to ensure continuity. As soon as text is edited in the sidebar, the rich-text JSON object is then applied back to the original object, ensuring a *sidebar* $\rightarrow$ *component* binding. This would cause the editor to reflect the new rich-text content.

Achieving the *component* $\rightarrow$ *sidebar* binding was easy, since the sidebar only pops up as soon as a component is selected, so the sidebar always gets the latest version of the text content. When editing text in the canvas, the sidebar is hidden.

7.6 Connector labels

A requirement of ours is that connector labels should be editable. However, we soon noticed that this was not so easy as we initially thought. DGM.js allows for connector labels snapped to every point of the connector, but this would not allow for distinguishing the *begin*, *middle* and *end* labels. In addition, we did not want to allow more than a begin/middle/end label and we wanted to configure which labels the user is allowed to set. This is why we opted to intercept the label-creation behaviour, in which labels snap to the begin, middle or end. In addition, since the label-creation code was altered, we added identifiers to the created Text components. Intercepting this creation behaviour also allowed us to check if creating a label on a position on the connector is allowed. For this, we created the `allowedLabelPositions` property for objects. In addition, some computed components required the connector to have a label which could not be edited, which is why labels can be set to read-only by the computed component. This can also be done via `allowedLabelPositions`. To make parsing of connectors even easier, the `parsedData` (from computedComponents) stores the labels.

7.7 General User Interface

For the user interface, fast templating needs to be supported, which is why we opted for a component library. Within front-end development, this is almost an industry standard. We chose PrimeVue, because of it's integration with tailwind, and our previous knowledge of the library. The best part of PrimeVue is that is has a so-called *Styled Mode*. In styled mode, PrimeVue builds upon Tailwind CSS, an industry standard popular class utility library. By using tailwind and PrimeVue, it allows us to design responsive, uniform user interfaces rapidly.

7.8 Design Decision: Fork or Patch

At the start of the project we relied on unedited source code of DGM.js. As we went along with our project we started to notice that we had to make adjustments to the editor core, and needed a connection with it. In the beginning these changes were minimal and small,

we used the plugin system that was provided by DGM, to register new shapes. Furthermore, we opted to change a few of the functions of the editor core by 'monkeypatching' them. Monkeypatching is the overriding of existing function logic in runtime. As our needs grew, monkeypatching became harder to maintain. Its reliability is questionable, because reloads or reinitializations could reset the function logic. There is also no syntax checking. Moreover, some of the functions from DGM that we wanted to use were not accessible.

First, we moved from monkeypatching to yarn patching for this. Yarn is the package manager that we use, and supports making patches that it will apply into the specified packages. This allows us to adjust DGM without having to fork the project. This worked pretty well, all functions were accessible, and there was no more runtime patching. After some careful consideration, we did finally move to forking DGM. Using patching was a mess, we had to edit the code in two big files, one for TypeScript type declarations and the other for the JavaScript code. These files also did not provide any syntax checking. Furthermore, it was hard to maintain, because every patch had to also include the changes of the previous patches. Because of this it was hard to see where what was changed.

Finally, we settled on setting up a fork of DGM. In GitLab there are two ways of doing this. The first one being, cloning the repository and manually pulling from the original project from time to time. The second one, setting up a GitLab Mirror. It will automatically copy branches from the original project and keep this up to date. We opted for the mirror because it would mean the maintainer would have to do less manual checking. It might also be possible to setup the automatic creation of merge requests from the mirrored branch to our production branch.

Forking is initially a bit more work to set up, as we now have two projects to take care of. Luckily, there is an easy way to combine these project into one working project by making it a monorepo. A monorepo is a repository which contains multiple projects. With Git and Yarn it is quite easy to set this up. First, we set up Git submodules, now we can assign a given git repository to a location in our workspace. We opened the UTML2 project and added our DGM fork as a submodule. We can still handle both git projects separately (visual studio code has good support for this). Now we can make a new Yarn Workspace to make sure that we depend on the DGM version which we are working on.

During the development of UTML, we also found some bugs within DGM.JS. For instance, once you want to snap a connector to an ellipse, it does not want to snap correctly. We have sent a bug report to inform the author, but we did not get a response to this. We decided to fix it ourselves, but were unable to suggest the changes to the author of the library, as they state that they do not accept pull requests.

7.9 Snappoints

Originally, DGM.js allows the users to connect a line to a shape at any point within the shape. For our use case this is unwanted behaviour. We adjusted the core to only allow snapping on the middle of the shape, edges, corners and, middle points of edges. It is configurable per shape which snapping options are enabled. For a more explicit way of snapping we also introduced a new shape 'SnapPoint'. Sometimes you need more explicit information about to which part of a diagram symbol a line is connected. It is already possible to check to what shape and where the line is connected, but it is tedious to keep track of. The SnapPoint is a very simple shape that can be added as a child of a bigger shape. Whenever a user drags a line, a suggestion box will show up where the user can snap the line to. Now it is very easy to check if a line is snapped to one or another point. This is useful for diagram symbols which need an input and have an output too, now you can easily distinguish how symbols are connected. To make sure SnapPoints stay

in the same relative position even if the parent shape changes size, we have made a new constraint. This 'scaling-anchor-to-parent' constraint has as input a relative point where the child should be positioned. For instance, `[0.1, 0.1]` will position the child 10% of the height down and 10% of the width to the right of the top-left corner.

7.10 Automata Arcs

In automata, you make transitions between states with an arc. This is an arrow that makes a rounded bow. At first we used the built-in curve linetype of DGM, which represents a Bézier curve. This worked quite well, but had terrible performance issues where users would get input lag from four or more Bézier curves. It also was tedious to move these curves, because the middle point which rounded the arrow had to be moved manually. The performance issues were the biggest problem. We tried to remove these by caching some of the calculations the curves were doing. Unfortunately this was no success. The main culprit for the issues seems to be DGM's updating pipeline. Under the hood it updates all shapes when one shape is updated. We took a different direction and tried to implement a new formula to replace the Bézier curves with. A simple parabolic formula did the trick, unfortunately, this does not work with well with the normal Connector implementation. In an original Connector implementation you can have an infinite amount of points on a path. As you can imagine it is not possible to make a path through this with a simple parabolic formula. Thus, we added a new Arc shape which extends Connector. The idea is simple, an Arc has three important values; the head, tail and controlpoint. The controlpoint is a point relative to the midpoint of the arc and determines the shape of the parabolic formula. Whenever you move one of the endpoints of the Arc, the controlpoint automatically moves to the correct point relative to the middle. This greatly improves performance and also user experience. We are now able to have at least 40 arcs in the editor without any lag, meaning that we have achieved an at-least 10 times improvement in the amount of curved connectors that can be on-screen.

7.11 Anubis Integration

Anubis is an exam environment at the University of Twente. For this project, an exam environment of the UTML editor had to be created. More specifically, we needed to create an HTML component that contains the full functionality of the editor, but can also be controlled using JavaScript from outside. To achieve this, we chose to compile UTML as an HTML webcomponent. This allows the entire application to be used as a separate HTML element, inside any HTML file. It bundles Vue and runs on raw JavaScript, so it does not require any special setup. To achieve this, we created a special build configuration. In it, we set environment variables, such that the application could detect that it is in webcomponent mode.

Since the webcomponent needs to execute on any (modern) browser, we had to bundle all frameworks that we are using, including, but not limited to Vue.js, PrimeUI, Tailwind and DGM.js. To ensure that most browsers can run our code, a compile target should be set. We use compile targets to compile our code to certain JavaScript language versions. Some of these versions are `ES6`, `ES2020`, `ES2022` and `ESNext`, with `ESNext` being the most recent specification of ECMAScript (which JavaScript implements), hence the `ES` prefix. We set the JavaScript compile target to `ES2020`, since it has good browser coverage [8]. It has all the features that are required for Vue. To minimize bundle size we mangled the

[8] According to , Caniuse.com, ES2020 has ∼95% browser coverage

JavaScript, however, we kept function names to expose our API to the webcomponent. An added benefit is that debugging is easier.

To expose some key functions, a new HTMLElement needs to be created, which specifies how an HTML should behave, which is needed for the browser to know how to use the new element. This element exposes some functions which can be interfaced with by Anubis, such as loading diagrams, toggling read only mode, or listening to events.

A large problem that we encountered was the styling within the webcomponent. The styling library (PrimeUI, building on top of Tailwind) assume global styling, but we only wanted styling to be affected inside the webcomponent. Essentially, at some point during development, the webcomponent worked and looked exactly like the non-webcomponent version, but the styling of elements inside the HTML container were also affected. Eventually, we had to include some CSS selector that would set everything that is not inside the `UTML2` to reset styling.

7.12 Testing

The system has mainly been tested with user testing interviews, and also system tests. For the user tests, we made a system that allowed us to do A/B testing. In the code we could switch between version A and B and also deploy these separately. Beside that we also implemented a few unit tests for Vue components in our UI with Vitest. For instance, the colour picker and zoom buttons are tested. We also chose to test the json files that contained the shapes of a given collection. There are some important requirements all shapes should have to function correctly and also so they can be parsed. For this we check whether all top-level json shapes have proto enabled and also if they have at least one tag.

We wanted to implement more extensive tests but this proved quite the challenge. The biggest part of our system is the editor that DGM.js provides. Unfortunately, this is largely untested within DGM.js. To test it ourselves is difficult, as it is difficult to test canvas elements in HTML. We have figured out that Vitest offers a browser mode, where it is possible to compare screenshots of the page. It might be possible to use this to test some interactions with our editor. However, this will likely be limited to the set of interactions that the browser testing allows for. Another approach would be to do some tests in the DGM.js repository directly. This would allow us to access more functionality of DGM.js easier. However, the DGM.js codebase is large, where we already had a hard time trying to understand it. Writing proper tests for this would take a significant while, and we considered it out of the scope of our project due to this. Another point is that we do not know precisely how certain functions or features are supposed to behave. We are dependent on the comments and documentation that is quite lacking in some aspects. Thus, writing tests does not mean we write proper tests. Of course, it would still be useful to test even though we are not certain about certain behaviour. This way we can test if the program keep behaving consistently after changing code.

Beside the testing that we do, we also have `ESLint` set up. This will check the code quality quickly and give suggestions in the form of errors and warnings. We have made sure that all errors have been patched, as well as a big chunk of the warnings.

In the pipeline the testing and ESLint is executed to check if there are no big issues.

7.13 Headless browser

A request of our supervisors was to have some automated way to export .utml files to images or vectors. Previously the supervisor did this process by creating a headless browser script

to automate the clicking of buttons. To implement this ourselves, we had to consider that exporting/rendering is done in-browser in DGM.js. This would only be possible in-browser, since the DGMjs editor is built in the HTML canvas, meaning that there are many elements relying on browser-specific or HTML-specific behaviour. An example of this, is that images in the Canvas are rendered using an `HTMLImageElement`, which is only available in-browser.

This means that creating a back-end or web API would be cumbersome, since it would have to render images in-browser. In addition, creating a back-end server would mean that we would have to add an HTTP server to our project for the single use of converting .utml files to images. This would add too much complexity, which is why we opted for creating a new automated browser script.

Using a script to click visible buttons is hard, since the script relies on the state of the UI. For instance, the menu collapses into a hamburger menu when viewing on smaller devices. Additionally, we would have to wait for the browser to render HTML elements, and then click on them, which is inefficient because we have to wait for the renders each time we click. This is why we exposed global JavaScript functions to the browser for importing and exporting files. This means that we could program our headless browser to call those functions directly, instead of clicking on buttons and having to wait on slow browser renders.

To make code interact with the browser, and to make the code easy to understand, we used the `selenium` package with python. Selenium is a well-known toolkit which is easy to install, easy to work with and manages the headless browser without extra code. To make the process of interacting with our code easier, we made use of the `argparse` library to provide an easy-to-use command-line interface with help menu.

7.14 Deployment & Pipelines

For UTML to be easily deployable, we opted to used Docker containers, since they are industry standard for easy deployment. For creating the container, we used a Dockerfile with two layers, a builder and a runner, such that only the build binaries of UTML (static static HTML/CSS/JS) and a minimal static file server would get shipped, reducing bundle size from roughly 1GB to 50MB. This large difference is caused due to the source code and all packages[9] are not included in the container.

Then, we setup two pipelines, one for testing and one for building the docker containers. The testing pipeline runs both the test suite and the linter only on merge requests or branch merges. The build pipeline is ran only on main and builds containers. The pipelines also allowed for easier development, since a merge request would only be merged if a pipeline has a successful build. It builds the latest commit and tags the container with the commit hash and the *latest* tag. This ensures that containers can always be rolled back to a certain commit hash, but the latest can easily be fetched with *utml2:latest*. In addition to the containers, we have also implemented A/B testing. Normally, A/B testing is done with a *campaign identifier*, and each campaign would define what experimental features to enable or disable. In addition, A/B testing for large-scale codebases includes the option to automatically enable it for a subset of the userbase. Since we did not need these features, we opted for an easier alternative, which we call the *alternate system*. The alternate system is built with an extra boolean environment variable, indicating if the current version is alternate or not. Then, the UI has some checks based on if it is the alternate version or

[9] Packages for browser development (`node_modules`) are known to take up a lot of space

not. During build, the alternate container is tagged using the commit hash combined with the alternate suffix. The most recent alternate version is available under the *alternate* tag, rather than the *latest* tag. This way, we were able to host the alternate version on a separate URL for user testing.

8 System Tests

To make sure our system works to intention we often conducted system tests. These are tests that are manually executed and compared to expected results. In Appendix E are the systems tests that we have performed. After implementing a feature, we also tested more specifically in edge cases for that.

One requirement is that the system should run on the chromebooks provided by the University during examinations. To test if this works correctly, we performed the system tests on chromebooks. The application worked well, except when ten or more Bézier curves were used. Then it would start to respond a bit slower. As a response to this we moved from Bézier Curves to Parabolic Arcs.

9 Risk analysis

9.1 Server risk

The application poses little to no risk for the system that it is running on, since UTML is a collection of static files served by a simple static file server, called *serve*[10]. Static files are safe, since there is no server-side code execution or direct access to system resources, There exists a risk of serve being hacked, but since it is a simple file server, it exposes no more secrets than would otherwise be exposed by the UTML binaries (which are loaded client side to access the application). In addition, UTML uses no secrets, so there lies no risk. Even if a hacker would obtain access to the file server, there is no risk to the server, since it is running in a docker container. To conclude, since the application is only serving static files, which is ran in a containerised environment, there is no risk to the server. In addition, serve is well-maintained.

9.2 User risk

The application runs only on the client side, in a container. Meaning that there are no risks associated to the user that cannot be solved with a simple refresh. However, that does not mean that an update of the application could not lead to errors. Since .utml files save the state of the shapes and components, updating the components themselves (like adding new logic, or changing the colour) would not update save files. This means, that if there were new updates added, that some save-migration system would have to be created.

9.3 Security risk

Since front-end projects tend to use many packages, supply chain attacks have become more popular recently. One of those attacks could lead to security risks. However, this would only take effect as soon as an insecure package is shipped. Another security risk would be that someone would obtain access to our static file server, which would lead to a compromise of all assets served to users. An attacker could inject their own code into the users' browser. However, the chance of the static file server being compromised is very low, since serve is a popular and well-maintained tool.

[10]Serve has ∼3 million weekly downloads. serve - npm

10 Reflection

In this section we will reflect on our working methods, reflect on the process and give recommendations for future work to be done on the project.

10.1 Organization

Since all of our team members live near the University of Twente, we decided to have daily meetings from Monday to Thursday. These meetings often started at 09:30 and lasted for about an hour. After this, most of the team would remain at the university, so we could discuss ideas and work together.

At the end of the week, after every meeting with our supervisors, we would have an extra meeting to discuss what we had learned and what the plan for the upcoming week would be. These meetings were led by Hanna or Sophie and minutes were taken so we could reflect. We think this system worked very well, as it kept everybody accounted for. This group worked better when we were all near another because ideas formed quicker. It also helped to keep everybody accountable because we expected them to show up every day unless they had a good reason not to be there.

At the beginning of the module, we made a planning, which we followed pretty well and made us stay on track. Quickly after the beginning, there formed a distinction between people within the group who were more inclined to implement, people who were more organizational, and people in between. This distinction got bigger as soon as we started to work on and researching the re-implementation of UTML, in which a knowledge gap got created. We were aware of this and tried to find tasks for everyone according to what they wanted to get out of the module. In a possible future, we might want to spend more time making all the group members more involved with the different aspects of this project, but overall we are content with how the task division went.

Our project was special because our supervisors were also our clients, namely a teacher and the future maintainer of UTML(2). This had some implications, because our supervisors are more involved with the project. It also meant that our supervisors had more reason to ask for features. During our meetings with the supervisors, we spoke to them as supervisors and as clients. Additionally, we involved our supervisor David, the future maintainer, more technically since he would need to understand what is going on. Because of this, we organised a technical meeting in which we spoke about the code structure and organisation.

10.2 Contributions

The roles were divided as follows:

- **Bram**
  Head programmer, in charge of code quality and testing, implemented a large part of the product. He was also in charge of the DGMjs fork and did a lot of research about the inner workings and how custom features could be implemented. Some notable implementations he has been responsible for are: snapping, SnapPoint shape, and Arc shape.

- **Daniel**
  Head programmer, picked out the libraries to use, chose the stack, and implemented a large part of the product. Daniel was also in charge of deployment and pipelines.

Also in charge of maintaining the issue board and ensuring that bugs/features would be tracked and divided into chunks. Responsible for two-way binding, computed components and Anubis integration.

- **Hanna**
  Team lead, led morning meetings or made notes during them. Kept up to date with everybody's progress and deadlines, made presentations for group sprints and colloquium. Created and sent out the student survey, Researched diagram types and made the list of requirements. Prepared and did user testing and teacher testing, compiled all feedback into usable documents, worked with programmers to translate this into GitLab issues. Sent out emails to all outside sources involved with the project, created the poster, in charge of report contents.

- **Sophie**
  Team lead, lead morning meetings or made notes during them. Made the agenda for supervisor meetings and in charge of the group calendar. Prepared and did user testing and teacher testing, In charge of report quality.

- **Twan**
  All-round programmer, who implemented lots of objects for the diagrams and made sure that all functionality was as what was expected. He also made sure that the dimensions and functionalities of the shapes are consistent between diagrams. Created the ethics form for the ethics committee and made the user information and consent form. In general, he helped out where needed (user testing, parts of the code, presentations).

10.3 General reflection

During our project, scope was a reoccurring issue. Sometimes, our ideas were out of scope, and sometimes we had our own doubts about features recommended by our supervisors. However, once our group realised this, we had frequent discussions of what the focus of our group would be and what would be in and out of scope. Throughout the entirety of our project we kept in conversation with our supervisors (who are also our clients) about whether things should be deemed inside or outside the scope.

For example, we wanted to add automatic evaluation of some diagram types, however we decided to scrap it since we did not have enough time to develop it. Something which we ended up implementing was two-way data binding and inline text editing, which was recommended to us by our supervisors.

Since we were faced with an already existing, outdated UTML, we are happy that we made the decision to re-implement it within the first week. This way we could quickly start on the implementation and made most use of the time we had. We think we chose good libraries for this project, even though there were shortcomings. DGM.js for example, does not allow contributions, requiring us to work around it (as written in *Technical Design*). In addition, DGM.js' license was hard to work with, there were long standing issues and the project had been stale for three months.

If we had more time, we would spend more time on designing the general layout and user experience. However, with frequent reflections on the UI throughout the entire process, we are positive that a good UI was designed. Later on we also did some user testing on elements we were uncertain about, which we relied on to make important UI decisions.

We have used Discord to communicate with our supervisors. We are happy with this arrangement, as it made it possible for the supervisors to stay up to date with our progress throughout the process, and allowed us to easily send them agendas for our meetings. This way we could also keep close ties with the future maintainer (one of our supervisors), allowing us to explain certain choices immediately and to reflect on and design certain features. Using Discord was a good choice, because it made communication more accessible than email due to it's casual nature.

Some feedback given by our supervisors is that we should keep track of where we are in the big picture, to take time to (re)consider decisions with a more clear mind. They felt that our fast implementation could be advantageous, but it could sometimes be too fast. Our supervisors also liked it that our group pro-actively tried to contact teachers to consult with them about the design and usability of UTML.

During our project, the new version of UTML was hosted on a server owned by one of our group members. It meant that the running version was easily accessible, which turned out to be very nice to have. Whenever we wanted to show off a new feature, we could simply refer them to the hosted website.

10.4 Future work

As we were working on this project, we encountered many bugs and also had many ideas. Unfortunately, our time is limited and therefore we chose to focus on the core features for the requirements. On the GitLab of the UTML2 project, as well as the DGMjs fork, we have an issue board. In the backlog there is a list of issues that we have chosen not to implement, but would be good features to have in the future. We will also suggest a few bigger suggestions for future work.

10.4.1 New diagrams

During the project we have done quite some research about the diagrams that were already in the old version of UTML. But, we also investigated implementing new diagrams of teachers that have showed interest in the previous version of UTML. One of those teachers came from the Electrical Engineering domain and they have interest in a diagram for circuit creation and also for bond graphs. Currently, the need for this to be implemented in UTML is not high, but it would be a good direction in making this tool accessible to more fields.

Another diagram that we did implement, but is still incomplete, would be the Fault Trees. In our design process, we discovered quite late that we missed some symbols in this. Due to time constraints, we chose to focus on polishing what we already had. It would be good to implement these in the future, so users are not confused why there are only a few symbols available to them for Fault Trees. Moreover, as earlier described in Technical considerations, Fault Trees could also have a better way of parsing the order in which the inputs of a gate are connected. This would make it easier and more fault proof for teachers to use.

10.4.2 Improving internal DGM.js update pipeline

One of our original requirements is that the new version of UTML should be able to have 500 shapes without noticeable lag. Unfortunately, we were unable to meet this requirement due to how DGM.js works internally.

During the project we got more understanding about DGMjs' functioning, which includes the rendering of the shapes, but also the resolving of constraints. For example, when one shape is moved in the canvas, it will update the constraints of all shapes in the canvas. Updating these constraints is an iterative process, in case the constraint is not met. The problem is that these constraint resolving updates are expensive to do. As you can imagine, having a lot of shapes can be problematic in this case. Another issue that amplifies this is that the editor does not have a set framerate. Because of this, the editor updates as soon as it can, leading to an unnecessary amount of updates which causes a computation queue. To illustrate, ideally every 1/60th of a second (one frame) you would execute updates for what has changed, merging all changes into one frame. This creates a set time to do these computations. Whereas, currently, for every small part of movement that you make, DGM.js updates immediately, causing it to update much more than 60 times per second. Due to this it can be very responsive at lower shape counts, but at the cost of performance.

In the future it would be wise to have more control of this update pipeline. First, by only updating the needed shapes. In many cases, from the kind of interaction, you can derive what needs to be updated. By only updating what is needed, the performance could improve with higher shape counts. It can be difficult to implement and test, due to unexpected updating behaviour it could introduce, since everything in DGM.js relies on this pipeline. Implementing a frame system is likely not realistic for UTML, though, since the core of DGM.js is quite convoluted. Many parts of DGMjs rely on browser events and are immediately executed. Converting this to a frame based system can be very difficult as it would require an overhaul of most code within the library.

10.4.3 Better automated tests

As mentioned in the technical design section, we do have a few automated tests, but not as much as we would have liked. We also described why some tests are difficult to do. However, it is still very feasible to, at least, test all the Vue components that are used within the user interface. Besides that, browser testing could work well if the screenshot solution introduced in technical design section works to our understanding. Then it would still be important to make sure the viewport and other system dependent settings are the same as in the screenshot. Moreover, to make the system even more reliable, unit tests could also be introduced to the DGMjs core.

10.4.4 Dark mode support

Currently, the logic for dark mode is already in place and all components also have the right dark mode colours. In some cases it even works precisely as expected. However, there are still a few bugs where the editor part of the interface is in light mode, and the other half in dark mode. Sometimes, a few parts do not want to change from light to dark or vice versa. It would be nice for user experience if this can be officially supported in the future by resolving these issues. This is likely an issue of our UI library, PrimeVue, interacting with Tailwind.

10.4.5 Diagram evaluation

During this project we also got forwarded some ideas that were originally sent to the maintainer. One of these ideas was to support the evaluation of automata. Our supervisor also seemed interested in this. After some back and forth conversations we considered

this out of scope for the project, because making an automaton parser is a whole different project. It is also already possible to parse the .utml file and to put the data into an external automata evaluator, such as JFLAP. Extending UTML2 to support evaluation would be possible, but requires structural changes. The diagram environment system would need to be expanded to support an Automaton environment, which would be selected by the user. Then, in this environment there is a given set of rules in place, such as having a single starting node and only using automaton symbols, to make sure the automaton can be evaluated. A new settings menu for evaluation should also be added. And then the most work of all, there should be a layer that can interpret the diagram as an automaton and then do different kinds of evaluation. Ideally, the evaluation would be extendable to different types of diagrams, and the maintainer should be able to very easily edit how evaluation is done.

10.4.6 Create an arrow from a shape

Within the DGM.js library there is the possibility of creating connectors from the edge of a shape. We did not implement this behaviour since we created our own computed connectors, which means that for each shape it is necessary to keep track of which connectors are relevant to that shape. While not impossible, this would take quite some refactoring of the DGM.js core and we did not have time for it within the timespan of this project.

10.4.7 Forward JSON compatibility

Both the old UTML and our new version UTML can import and export `.utml` files. However, even though the files share the same filetype, the way the JSON content is formatted is very different. Because of this, it is currently not possible to import old UTML files into the new version. One teacher indicated that it could be desirable to have forward compatibility. Thus, in the future it could be a possibility to see if old diagrams can be converted into new diagrams of our system. This would mean that a translation layer between old and new UTML needs to be written.

10.4.8 Anubis testing

While we did create an Anubis integration, there was unfortunately not enough time to properly test it in the live Anubis deployment. We did reach out to the relevant teacher, but they did not reply to us at the time of writing (which we assume is partially due to the Anubis program crashing during an exam, the week before our project deadline), so this would still have to be done before UTML can be used in Anubis.

A Survey results

Complaints and bug reports	Count
The cursor selects at a distance sometimes	15
The "help" box on the left side of the screen gets in the way	8
Student cannot load a saved .utml file	7
The system starts to lag when working on a big file	5
Classes disappear (after zoom)	4
Leaving the page resets your diagram without saving and without warning	3
The Delete button on a keyboard deletes an object, not the text (when typing)	3
Zoom makes visibility hard	3
Student cannot save in cache	2
Class is secretly one big text box (old tool)	2
Object list disappears	1
Some diagrams don't look the same when exported	1
You cannot scroll up or to the left	1
The direction of links is inverted in sequence diagrams	1
The "alternative" object should have 2 boxes instead of one	1
Copying makes text disappear	0

Table 2: User Feedback Negative

Feature Requests	Count
The grid should be optionable or have more positions to snap arrows to.	14
There should be more snapping points to the sequence diagram execution bars (currently 5)	6
Multiselect would be nice to have	3
You have to manually put in the offset of the alternative box in the sequence diagram. Having the option to drag it would be a lot nicer	2
It would be nice if there was a boilerplate for a specific diagram	2
It would be nice if you could add different colours to objects	2
It would be nice if you could change the type of background (transparent, white, black) on export	1
The toolbox should be able to be toggled on or off	1
Add text to the object in the toolbar instead of double clicking	1
It would be nice if you could change the type of arrow after adding it	1
Maybe the system could become Open Source	1
It would be nice if keyboard shortcuts were better supported	1
Currently the tool is missing Entry/exit activities for State Machine Diagrams	1
It would be nice if copying arrow text also copied the arrow	0

Table 3: User Feature requests

Best liked features	Count
The software is simple and intuitive	8
The drop-down menu with different diagrams is very useful	8
The history feature in the new version	4
Drag and dropping	2
Export as different types	2
Being able to edit diagrams directly using the XML file	1

Table 4: User Feedback Positive

B User stories

B.1 Student: Sytse

Sytse is a student who, for his module 2 design subject, has to make a class diagram of his software. For this he uses the software provided by the UT, UTML. He opens UTML, selects the Class Diagram package and starts working. He places a class object, but gets stuck on how to use the arrows, so he selects the help button. He combines two class objects with an arrow on two snapping points he chose. He quickly changes the text in one of the class objects and adds a new method. He changes the classes around and the arrows still look nicely. Then he downloads the class diagram to a png and uploads it to canvas. He also saves it as UTML, so he can change anything later. Later in the modules he wants to change the class diagram a bit, so he opens his .utml file, makes his changes, and exports it again.

B.2 Student: Sterre

Sterre is a student who has to take a test. She opens Anubis and eventually comes to a point where she has to make an activity diagram. Within Anubis UTML is embedded in a box under the question where Sterre can make the diagram. She can only use elements from the activity diagram and can do some other things. Buttons such as importing and exporting files have been replaced with a save button. She makes her diagram and presses the save button. Later, she looks over her answers again and realises she wants to change a couple of things. She makes the changes, presses save again, and finishes the test.

B.3 Student: Stephan

Stephan is a student following the Languages and Machines course. For this course he has to make several automata, that all can parse different languages. He creates the diagram. Then he goes to the built in terminal and runs the automata with the parameters specific to the diagram. He gets an error message saying it does not yet pass the language correctly, and shows an example of where it goes wrong. Stephan changes the automaton to now parse the language correctly, goes to run it again, and gets a "Language parsed" message.

B.4 Susan

Susan is a teacher for the module 2 design course. She wants that students can recreate the example diagrams showcased in the module guide within the UTML application. At the end of the course, the student will have to take an exam where they need to create a class diagram. Susan would like to get all .utml files that she can easily parse. When a mistake is noted by her parse, she would like to double check. For this, she converts the .utml files to .svg in large batches in UTML. After the test, Susan gets informed by students that there is a mistake in one of the diagram symbols. Susan contacts the maintainer and the issue is quickly corrected.

C Diagram requirements

Since the key feature of UTML is that users can create diagrams with the software, we decided to dedicate this section on what diagram types should be supported and what features they should have. These are only the base diagram types; teachers can always add their own diagram types later. We based these diagrams on the options currently already implemented in UTML. The first 5 (Activity, Class, Use Case, Sequence and Automata) have priority, as they are used more and have the highest barrier to implement.

C.1 Activity Diagram

The system should be able to support the creation of activity diagrams with the following attributes:

- Start: a filled in circle with 4 snapping points to mark the start of the diagram.
- End: not-fully filled in circle with 4 snapping points to mark the end of the diagram
- Swimlane: a box with dotted lines, a text field at the top for the title. Used in the background of the diagram to showcase which actor does which activity.
- Activity: a box with one text field and many snapping points. Used to describe an activity.
- Fork/Rejoin: A filled in bar with many snapping points. Used to split or join arrows
- Decision Node/merge node: A diamond shape with 4 snapping points to add conditions to arrows or to merge 2 arrows.
- Arrow: connectors between the multiple other objects, should have an optional text field to add conditions.
- Hourglass: an hourglass shape with multiple snapping points, used to showcase a "waiting" state.

Optional extra features: because an activity diagram can only have 1 start point, we could try to restrict the user to only be able to add one.

C.2 Class diagram

The system should be able to support the creation of Class diagrams with the following attributes:

- Class: a box with 3 fields. The title field, attributes field and operations field. The title field is mandatory; the other 2 fields are both optional. The attribute field should be able to list multiple attributes (called fields in the current version of UTML) and the operations field should be able to hold multiple operations (called methods in the current version of UTML).The entire class object should have multiple snapping points for arrows.

- Connectors: There should be 6 different kinds of connectors in the class diagram: Association (normal line, no arrow head), Inheritance (normal line, triangular empty arrowhead at the target), Realization (dotted line, triangular filled in arrowhead at the target), Dependency (dotted line, 2 lined arrow at the target), Aggregation (normal line, empty diamond at the origin) and Composition (normal line, filled in diamond at the origin). (See image below) These connectors should have 3 optional text fields, one at the origin, one at the middle and one at the target.

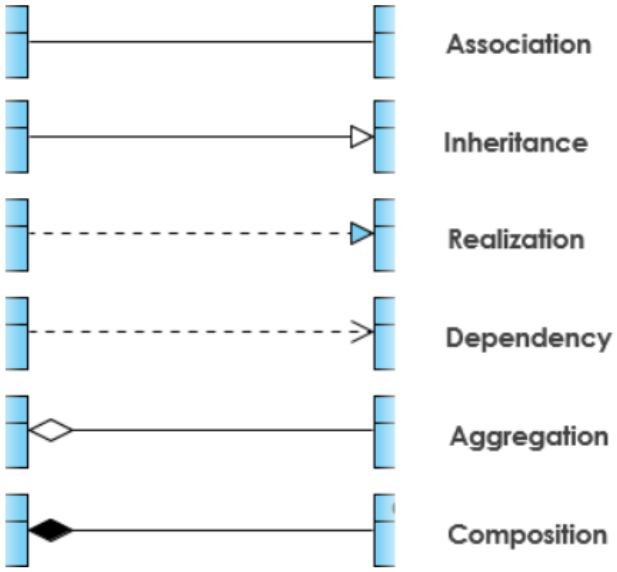


FIGURE 10: Class diagram connectors

Optional extra features: for a clean look it would be nice if connectors could merge, so 2 classes could point to another class with only one connector. This however seems difficult to implement, and is of low priority.

C.3 Use Case Diagram

The system should be able to support the creation of Use Case Diagrams with the following attributes:

- Actor: The actor is represented by a stickfigure with multiple snapping points for connectors. There can be multiple actors in one diagram, each Actor must be linked to a use case.
- Use Case: A use case is an oval with text in it. Should have multiple snapping points.
- Connector: There should be 4 types of connectors supported: a regular communication link (solid line), Extends (dotted line with 2 lined arrow head and the text «Extend» above it), Include (dotted line with 2 lined arrow head and the text «Include» above it) and Generalization (solid line with triangular emtpy arrow head).
- System boundary: a rectangular box with a title field. Should be in the background of the diagram, needs no connectors.

Optional extra features: It would be nice if the Extend connector adds the extended child use case title in the parent use case. (See image below)
The current version of UTML also has a "System Clock" feature; nowhere in the documentation of Use Case diagrams can we find out what this is needed for.

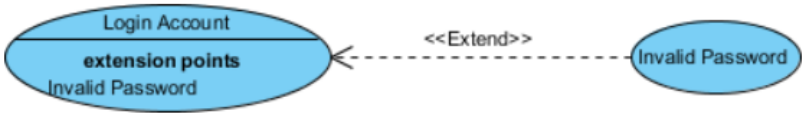


FIGURE 11: Extend in a use case diagram

C.4 Sequence Diagram

The system should be able to support the creation of Sequence Diagrams with the following attributes:

- Actor: The actor is represented by a stick-figure, there can be multiple actors in one diagram. Each actor should have one snapping point at the bottom for an execution.
- Lifeline: a rectangular box with a text field in it and with a dashed line at the bottom. There can be multiple Lifelines in one diagram. The dashed line should be able to have varying length.
- Executions: Thin rectangles on the dashed lines of the lifelines, the top and bottom represent the initiation and completion time of an operation. These rectangles should have many snapping points on their left and right sides for messages.
- Messages: a message is an arrow connector from one lifeline to another. There should be multiple message options: Call messages (solid line, two-lined arrowhead signaling direction), Return Messages (dotted line, two-lined arrowhead signaling direction), Self message (solid line, two-lined arrowhead, going from one lifeline to the same lifeline)
- Destruction: a big X object marking the end of a lifeline
- Alt: a rectangular box with the text "Alt" in a label in the top left corner. Should be able to have multiple (optional) "condition" fields. The Alt box should be able to go behind parts of the diagram and needs no snapping points.
- Loop: a rectangular box with the text "loop" in a label in the top left corner. Should be able to have 1 condition field. Also goes behind parts of the diagram and needs no snapping points.
- Sequence Control Flow: a rectangular box with the text "Opt" in a label in the top left corner. Should also be able to have 1 condition field. Also goes behind parts of the diagram and needs no snapping points.

Optional extra features: The current version of UTML makes users add lifelines and their corresponding dotted lines separately. Since every lifeline must have a dotted line we could maybe merge them into one object.

C.5 Automata

The system should be able to support the creation of Automata with the following attributes:

- State: A state is a circle with a text field. A state should have multiple snapping points.
- End State: An end state is a circle with a double outer line. (so a circle in a circle). With a text field and multiple connection points. There can be multiple end states in one Automata
- Connectors: The connectors in an automata are arrows (solid lines, regular arrowhead), they can be straight or curved. They should be able to curve to the point of pointing to their original state. These arrows should have a text field in the middle.

Optional extra features: In the current version of UTML is an "Init Arrow" A connector with a filled in diamond shape at one end, unclear what this is used for, but it should be easy to add as an option.

C.6 Additional Diagrams

After these basic diagram types have been implemented, the following diagrams will also be added. They are less of a priority as they are less used, and are also easily implemented after the first 5 are, since they reuse components already created.

State Machine Diagrams: The system should be able to support the creation of State Machine Diagrams. These are a lot like class diagrams in the sense that they have the same Start, End, Fork/Rejoin and basic Arrows. The difference is that the state machine diagram has States: rectangular boxes with multiple snapping points and one text field. And Compound States: rectangular boxes with a title field and a text field, also with multiple snapping points. Another difference between a state machine diagram and a class diagram is that a state machine diagram can have multiple start points.
Derivation Trees: The system should be able to support the creation of Derivation Trees, these Trees consist of only Nodes: rectangular objects with a text field and many snapping points. And Edges: solid lines connecting the Nodes.
Fault Trees: The system should be able to support the creation of fault trees. These trees consist of Basic events: circles with a text field and multiple snapping points. Intermediate events: rectangles with a text field and multiple snapping points. Edges: solid lines connecting the events. And lastly “And-” and “Or-gates”. These objects should have multiple snapping points in order to be able to join edges and have an outgoing edge.
Extended Automatas: This diagram has all the features of the previously mentioned automata, and next to that, an “Edge split off node” a black, filled-in circle with multiple snapping points. And rectangular Nodes with text fields (in addition to the circular nodes)
Computer Architectures: A computer architecture consists of only rectangles with text fields. These should be very easy to add, but are also of low priority.
Graphs: The graphs section of the current version of UTML just has some edges, curved edges and square and circular nodes. To ensure the user can still make Graphs in the updated version of UTML we will have a “Sandbox mode” where the user has access to all objects across all diagrams.

D Testplan

Study Background
This study aims to create UTML2, a web-based tool based on the original UTML made by the University of Twente. Both UTML2 and UTML are used to create UML diagrams. The purpose of this test is to check the functionality and usability of the tool, and identify possible bugs. The collected results will be used to (re)design the application and to motivate certain design choices.

What you are being asked to do
You are asked to complete an exercise of the M2-Software Systems - Design course using UTML2. Please try to complete the exercise to the best of your ability, but there are no consequences of making mistakes or not completing the exercise. You are asked to say your thoughts out loud when you are doing the exercise. After the exercise you will be asked some general questions by the researcher about your experience using the tool. The test will take between 10 and 20 minutes.

What are the risks and benefits
There are no known risks associated with participating in this test beyond possible minor inconvenience from using the tool and answering the questions. While there are no direct benefits to you as a participant, your feedback will help improve the UTML2 website for future users.

Data collection and usage
During the test the researcher will take notes of what you say and do. All results will be handled with care and analyzed anonymously. No personally identifiable information will be used to identify you in any reports or publications. The collected data will be deleted 5 weeks after the date of testing.

Your right to withdraw and withhold information
Participation in this test is entirely voluntary. You may withdraw from the test at any time by telling the researcher, without any negative consequences. You are not required to perform tasks or answer questions you do not wish to do. Any data you provide may be withdrawn before the data analysis phase by contacting the researcher. For questions about this study, please contact: t.weerdenburg@student.utwente.nl.

By signing this form you agree to have read and understood the mentioned above.

Name of participant:
Signature:

_____

Location:
Date:

E System Tests

Identifier Legenda

CP = Component Panel	TB = Toolbar	PP = Property Panel	SP = Structure Panel
SS = Session Storage	TE = Text Editing	IE - Import/Export	MM= Main Menu
ED = Editor General	EM = Editor Movement	EB = Editor Basic Shapes	EC = Editor Custom Shapes
AI = Anubis Integration			

Identifier	Description	Instructions	Expected Output	Actual Output	Pass/Fail	Remarks
CP1.1	Drag and drop of diagram symbols	1, open components 2, open comments collection 3, click and drag on comment into the editor	There should be a comment in the canvas where the cursor points to the middle	There is a comment in the canvas where the cursor points to the middle	Pass	
CP1.2	Double click diagram symbols	1, open components 2, open use case diagram 3, double click on Use Case	There should be a Use Case in the middle of the open editor	There is a Use Case in the middle of the open editor	Pass	
CP2.1	Searching for a diagram symbol	1, open components 2, search for “link”	three collections; Comments, Class Diagram, and Use Case Diagram should show up all with one symbol that has link in its name	Comments, Class Diagram and Use Case Diagram show up, all with one link symbol in it	Pass	
SP1.1	Dragging multiple elements in shows up in structure panel	1, repeat CP1.2 3x 2, open structure panel	Use Case appears three times under structure	Use Case appears three times under structure	Pass	
SP2.1	Swimlane with multiple elements shows up in structure panel	1, repeat CP1.2 3x 2, add swimlane under activity diagram 3, select all use cases by clicking and dragging 4, drag them into the swimlane (see blue edge)	Swimlane appears in structure and can be unfolded to reveal three use cases	Swimlane appears in structure and can be unfolded to reveal three use cases	Pass	
SP2.2	System boundary within a system boundary with elements shows up in structure panel	1, repeat SP2.1 but with system boundary instead of swimlane 2, make the existing system boundary bigger by grabbing an edge and dragging 3, add a new system boundary and drag that into the other system boundary 4, repeat CP1.2 and add it to the small system boundary	A system boundary in structure which can unfold. Which then reveals three use cases and also a system boundary that also contains a use case.	A system boundary in structure which can unfold. Which then reveals three use cases and also a system boundary that also contains a use case.	Pass	
MM1.1	Opening of Help menu, and using tabs	1, click help button 2, click on movement, and the other tabs 3, click outside of it to close	A pop-over should appear over everything else on the screen. When clicking other tabs it switches the text. When clicked outside it closes.	A pop-over appears over everything else on the screen. When clicking other tabs it switches the text. It closes when clicked outside.	Pass	
MM1.2	Opening shortcuts tab	1, click shortcuts button	A pop-over should appear over everything else with most shortcuts.	A pop-over appears over everything else with most shortcuts.	Pass	
MM1.3	Clicking go to origin icon	1, click the round-ish target icon “go to origin”	The origin indicator should be in the middle of the editor	The origin indicator is in the middle of the editor	Pass	
MM2.1	Toggle grid setting	1, click settings button 2, toggle grid off 3, toggle grid on	Grid should be first on. After toggling it off it should disappear and the origin indicator too. After turning it on again both appear.	Grid is on first. After toggling off, the grid disappears but the origin indicator stays. After turning it on the grid comes back.	Fail	Can be changed easily.
MM2.2	Toggle snap to grid, also updates with keybind	1, click settings button 2, toggle snap to grid on 3, Repeat CP1.2 4, Drag shape around 5, do ctrl+g 6. drag shape around	Snap to grid is default off. After it is turned on, the shape snaps to grid when moved. When off, it is not snapped to grid anymore.	Snap to grid is default off. After it is turned on, the shape snaps to grid when moved. When off, it is not snapped to grid anymore.	Pass	
MM2.3	Toggle snap to object, also updates with keybind	1, click settings button 2, toggle snap to object off 3, Repeat CP1.2 2x 4, Drag shape beside other shape 5, do ctrl+o 6. drag shape around	By default snap to object is on. After toggling off the shapes do not show any snapping behaviour to another with blue lines. After turning it on it does.	By default snap to object is on. After toggling off the shapes do not show any snapping behaviour to another with blue lines. After turning it on it does.	Pass	
MM2.4	Toggle show zoom	1, click settings button 2, toggle zoom button off 3, toggle it on	By default it shows the reset, zoom-in and zoom-out buttons. After toggling off they disappear, and return when turned on again.	By default it shows the reset, zoom-in and zoom-out buttons. After toggling off they disappear, and return when turned on again.	Pass	
TB1.1	Clicking Selection	1, click on the cursor icon in toolbar 2, repeat CP1.2 2x 3, drag one of the shapes away by clicking on it and dragging. 4, click and drag on an empty canvas point over both shapes	By default the selection tool is selected. After dragging one of the shapes, it should be moved to the new position where the cursor let it go. Dragging over multiple shapes selects them.	By default the selection tool is selected. After dragging one of the shapes, it should be moved to the new position where the cursor let it go. Dragging over multiple shapes selects them.	Pass	
TB1.2	Clicking hand	1, click on the hand icon in toolbar 2, click on screen and drag	The canvas should have moved in the opposite direction the user was dragging	The canvas moved in the opposite direction of dragging	Pass	

TB1.3	Clicking undo and redo	1, repeat CP1.1 2x 2, press Undo button toolbar 3, press Redo button toolbar	Two shapes are first added, then the last added shape is removed and added back again.	Two shapes are first added, then the last added shape is removed and added back again.	Pass	
TB1.4	Click zoom in and out button	1, click on zoom in button toolbar 2, click on zoom out button toolbar	The origin indicator and grid should appear bigger, then should appear smaller again but same size as at start	The origin indicator and grid should appear bigger, then should appear smaller again but same size as at start	Pass	
TB1.5	Reset zoom	1, click zoom in 5x 2, click reset zoom	The origin indicator and grid should appear the same as before zooming in	The origin indicator and grid are the same size after clicking reset	Pass	
TB1.6	Duplicate button enabled and click it	1, repeat CP1.2 2, select shape 3, click duplicate button	when no shape is selected the button is disabled, after selecting it becomes enabled. After clicking it, it duplicates the shape.	when no shape is selected the button is disabled, after selecting it becomes enabled. After clicking it, it duplicates the shape.	Pass	
TB1.7	Delete button enabled and click it	1, repeat CP1.2 2. select shape 3, click remove button	when no shape is selected the button is disabled, after selecting it becomes enabled. After clicking it, it deletes the shape.	when no shape is selected the button is disabled, after selecting it becomes enabled. After clicking it, it deletes the shape.	Pass	
IE1.1	Exporting as UTML, and importing	1, Repeat EM2.1 but with utml instead of png. 2, Click on file button (top-left) 3, Click on new file 4, Click on new file in pop-up 5, Click on file button 6, Click on import 7. Click on choose and select the exported file.	The imported file shows the same canvas as the exported file.	The imported file shows the same canvas as the exported file.	Pass	
IE2.1	Exporting as PNG	1, repeat EM2.6 2, click on File button (top-left) 3, click on export 4, select png 5, put test in filename 5, click on download	The output shows all objects like in the fitted view with a white background	The output shows all objects like in the fitted view with a white background	Pass	
IE2.2	Exporting as SVG	Repeat IE2.1 but select svg instead of png	The output shows all objects like in the fitted view with a white background	The output shows all objects like in the fitted view with a white background	Pass	
PP1.1	When nothing is selected the property panel shows nothing	1, click on the round icon to the left off the screen to open the property panel	It should show that you need to select something first	It shows that I first need to select something	Pass	
PP1.2	The property panel is automatically expended when a text-editable or property editable shape is selected	1, Repeat CP1.2 but for class diagram 2, click on it to select it 3, Repeat CP1.2 but for Decision/Merge 4, click on it	The property panel should automatically expand from the left. When Decision/Merge is clicked is collapses	The property panel expends automatically from the left. When Decision/Merge is clicked is collapses	Pass	
PP1.3	Editing text does not show up when multiple shapes are selected	1, Repeat CP1.2 2x 2, Select selection button in toolbar 3, drag over the shapes	The sidebar does not show up	The sidebar does not show up	Pass	
PP2.1	Editing text in the property panel changes text in editor	1, Repeat CP1.2 2, Change the text in the left sidebar to “Test”	In the editor the use case should now have Test instead of Use Case	In the editor the use case should now have Test instead of Comment	Pass	
PP2.2	Editing inline text changes text in property panel	1, Repeat CP1.2 2, Change the inline text to “Test” 3, Reselect the comment	In the sidebar it should now have Test instead of Use Case	In the sidebar it should now have Test instead of Use Case	Pass	
PP3.1	When a property is adjusted, it reflects in the editor	1, Repeat CP1.2 for class diagram 2, select class diagram 3, click + under fields 4, enter Test in name	In the class there now is - Test	In the class there now is - Test	Pass	
PP3.2	When a property is edited multiple times in one go it does not break	1, Repeat CP1.2 for class diagrams 2, select class diagram 3, click on “Class” under class type, 4, change it 5x to another value	Whenever the class type is updated it immediately reflects it in the editor with <<abstract>> or <<interface>> or nothing.	Whenever the class type is updated it immediately reflects it in the editor with <<abstract>> or <<interface>> or nothing.	Pass	
PP3.3	Shape properties do not show up when multiple shapes are selected	1, Repeat PP1.3, but for class diagrams	The sidebar does not show up	The sidebar does not show up	Pass	
SS1.1	Pressing ctrl+s saves your progress	1; Open two instances A and B in tabs 2; Start fresh on instance A, don’t click on anything on B. 3; A: Load components into editor 4; A:CTRL+S 5; B: Refresh sessions button and select A session to see update	Newly added objects should be visible in B session screen	Newly added objects are visible in B session screen	Pass	Not in Anubis
SS1.2	Without pressing ctrl+s it saves your progress	1; Open two instances A and B in tabs 2; Start fresh on instance A, don’t click on anything on B. 3; A: Load components into editor 4; A:Wait 30s 5; B: Refresh sessions button and select A session to see update	Newly added objects should be visible in B session screen	Newly added objects are visible in B session screen	Pass	Not in Anubis
SS2.1	Naming session	1; Open two instances A and B in tabs 2; Start fresh on instance A, don’t click on anything on B. 3; Rename session A 4; B: Refresh sessions button	Session name of A is changed	Session name of A is changed	Pass	Not in Anubis

SS3.1	Session list shows up after reload	1; Open editor 2; add object 3; Reload page	Session list shows	Session list shows	Pass	Not in Anubis
SS3.2	Session list shows up after opening page	Repeat SS3.1, but instead of reloading, open a new page.	Session list shows up	Session list shows	Pass	Not in Anubis
EM1.1	Ctrl + (+/-) zooming	1, press ctrl + (+) 2, press ctrl + (-)	The editor should zoom in and then out to its original zoom level	The editor zooms in and out to its original zoom level	Pass	
EM1.2	Ctrl + scroll zooming	1, hold ctrl and scroll up 2, hold ctrl and scroll down	The editor should first zoom in and then zoom out according to how much you scroll.	The editor should first zoom in and then zoom out according to how much you scroll.	Pass	
EM1.3	Use trackpad zooming	1, use a trackpad 2, unpinch your fingers 3, pinch your fingers	The canvas should first zoom in and then out again	The canvas should first zoom in and then out again	Pass	
EM2.1	(shift+) scroll movement	1, scroll up and down 2, scroll up and down while holding shift	First it will move vertically through the canvas depending on your scrolling. Then it will move horizontally through the canvas	First it will move vertically through the canvas depending on your scrolling. Then it will move horizontally through the canvas	Pass	
EM2.2	Middle click or spacebar + drag movement	1; Open editor 2; Hold middle click and drag 3; Hold spacebar and click and drag	Both options should move the editor	Works!	Pass	In Anubis, on Windows, middle-click scrolling triggers on top of this mode
EM2.3	ctrl + arrows movement	1; Open editor 2; Ctrl+left arrow 3; Ctrl+rightarrow 4; Ctrl+up arrow 5; Ctrl+down arrow	Each arrow press should move the canvas in that direction	Canvas is moved appropriately	Pass	
EM2.4	Use trackpad scrolling	1, use a trackpad 2, with two fingers move up and down 3, with two finger move sideways 4, with two fingers move diagonally 5, with two fingers in circles	The canvas moves in the opposite direction vertically. The in the opposite direction horizontally. Then both are combined and the canvas moves diagonally. The canvas moves in circles	The canvas moves in the opposite direction vertically. The in the opposite direction horizontally. Then both are combined and the canvas moves diagonally. The canvas moves in circles.	Pass	Might be restricted by the hardware of the trackpad
EM2.5	Ctrl + 0 reset zoom	1; Zoom around the editor 2; Ctrl+0 3; Zoom once more 4; Reset zoom button	In both cases, zoom should reset	Zoom resets	Pass	
EM2.6	Ctrl + 9 fit zoom	1, Repeat CP1.2 4x 2, select a shape and move it quite a distance away 3, repeat for all shapes 4, press Ctrl + 9	The editor should zoom out and position itself such that you can see all elements on the canvas	The editor zooms out and positions itself such that you can see all elements on the canvas	Pass	
ED1.1	Ctrl + z (undo) & Ctrl + y or Ctrl + shift z (redo)	1, Repeat CP1.2 2, Ctrl+ Z 3, Ctrl + Y 4, Ctrl+Z 5, Crtl + Shift + Z	Use Case should appear, than disappear when Ctrl+Z, then appear when Ctrl+Y then disappear when Ctrl+Z and appear when Ctrl+Shift+Z is pressed	Use case appears first. Undo and redo work fine.	Pass	
ED1.2	Ctrl + c (copy) & Ctrl + v (paste)	1, Repeat CP1.2 2, select the use case 3, ctrl+c 4, move to another point 5, ctrl+v	The use case is copied at the middle of the screen at the other position	The use case is copied at the middle of the screen at the other position	Pass	
ED1.3	Ctrl + x (cut) & ctrl + v (paste)	Repeat ED1.2 but with ctrl+x instead of ctrl+c	The use case is copied to the new location in the middle of the screen and disappears from the old location	The use case is copied to the new location in the middle of the screen and disappears from the old location	Pass	
ED2.1	Del or backspace to delete	Repeat TB1.7 2x but instead of pressing the button, press backspace and delete	The added shape is removed	The added shape is removed	Pass	
ED2.2	moving shapes with arrow keys	1, Repeat CP1.2 2, select the shape 3, press all arrow keys after another 4, press all arrow keys + shift	the shape moves respectively to the arrow direction, when shift is pressed it moves a bigger distance	the shape moves respectively to the arrow direction, when shift is pressed it moves a bigger distance	Pass	
ED2.3	bring forwards, backward or to front	1, Repeat CP1.2 3x 2, Move the shapes to overlap on another 3. Select the bottom shape, 4, Press] 5, Press [6, Press ctrl +] 7, Press ctrl + [	First the selected shape will go up, then it goes back again. Then it goes to the top, and then it goes to the bottom again.	First the selected shape will go up, then it goes back again. Then it goes to the top, and then it goes to the bottom again.	Pass	
ED2.4	Toggle snapping with keybinds	Most is tested in MM2.2 & 2.3 1, go to settings 2, toggle grid snap and object snapping on 3, Repeat CP1.2 2x 4. select a shape and drag beside the other 5. repeat while holding ctrl	ctrl+o toggles object snapping, ctrl+g toggles grid snapping. Holding ctrl disables snapping.	ctrl+o toggles object snapping, ctrl+g toggles grid snapping. Holding ctrl disables snapping.	Pass	

ED	Moving a diagram symbol	1, Repeat CP1.2 2, Select the selection (cursor) toolbar tool 3, click and drag the shape to move around	The symbol should move with your cursor	The diagram symbol moves with the cursor	Pass	
ED	Resizing a diagram symbol	1, Repeat CP1.2 2, Select the selection (cursor) toolbar tool 3, Select the shape 4, click and drag on a square at the outline of the shape.	The symbol should be resized in the respective direction	The symbol is resized in the respective direction	Pass	Not all shapes support resizing, or can only be resized in a specific direction.
ED	Connecting to an ellipse outline works	1, Repeat CP1.2 for Lost/Found 2, Repeat CP1.2 for Comment Link 3, select the comment link 4, click and drag the endpoint of a comment link onto the Lost/Found	The endpoint of the comment link should only snap on the outline of the Lost/Found and the middle point. No other points.	The endpoint of the link snapped only on the outline and middlepoint, nothing else.	Pass	
EB	Moving an arrow-type	1, Repeat CP1.2 for comment link 2, select the shape 3, click and drag on the shape to move it around	The symbol should be moving to the cursor, but not changing its path	The symbol is moved to the cursor, but does not change its path	Pass	Internally, moving of arrows works differently than most shapes
EB	Moving the endpoint of an arrow type	1, Repeat CP1.2 for comment link 2, select the link 3, click on an endpoint circle and drag it to another point (or shape)	The link should change its path based on the endpoints, the other endpoint does not move.	The dragged point moves to the cursor, but the other point stays the same. The path of the link is updated.	Pass	
EB	Adding points to an arrow- type	1, Repeat CP1.2 for comment link 2, select the link 3, click and drag on the + circle between existing points.	The endpoints stay in the same place, a new point is added at the place of the cursor and the path goes through this point.	The endpoints stay in the same place, a new point is added at the place of the cursor and the path goes through this point.	Pass	
EC1.1	Moving an Arc	1, Repeat CP1.2 for Arc 2, select the shape 3, click and drag on the shape to move it around	The symbol should be moving to the cursor, but not changing its path	The symbol should be moving to the cursor, but not changing its path	Pass	
EC1.2	Moving the control point of an Arc	1, Repeat CP1.2 for Arc 2, select the arc 3, click and drag on the white circle near the middle of the arc	The parabolic path of the arc should be changed to go towards the controlpoint	The parabolic path of the arc is changed to go towards the controlpoint	Pass	
EC1.3	Moving an endpoint of an Arc	1, Repeat CP1.2 for Arc 2, select the link 3, click on an endpoint circle and drag it to another point (or shape)	The dragged endpoint is moved to the cursor, the control point is moved so that it is still in the same position relative to the middle. The other endpoint stays in the same position	The dragged endpoint is moved to the cursor, the control point is moved so that it is still in the same position relative to the middle. The other endpoint stays in the same position	Pass	
EC2.1	Double clicking on arrow makes a new label at correct position	1; Drag class diagram connector into editor (all position are allowed) 2; Double-click on begin of connector 3; Repeat step two for the middle and end	All textboxes should be created on a set position on the arrowe	All textboxes are in the correct position	Pass	
EC2.2	Double clicking on arrow with labels, selects the right label	1; Repeat EC2.1 2; Click on the connector in all three positions (begin, middle, end)	Text shapes should get selected	Correct corresponding text shapes get selected	Pass	
EC2.3	Arrows with a readonly label, cannot have this arrow removed or edited	1; Drag Include connector from Use Case into the editor 2; Try to edit label	Label is not editable or removable	Label is not editable or removable, even if the text shape is selected and delete is pressed	Pass	
EC	Snapping to a snappoint works	1; Drag in snappoint 2; Snap line to it	Line snap	Line snap	Pass	
EC	Alt box gets new boxes when a new item in property panel is added	1; Drag in Alt component 2; Change condition text 3; Add 1 fragment item 4; Remove fragment item 5; Add 2 fragment item	New boxes are shown	New boxes are shown	Pass	
EC	Alt box size can be adjusted	1; Drag in Alt component 2; Change condition text 3; Add 1 fragment item 4; Change text of one fragment in sidebar 5; Change text of other fragment inline 6; Adjust size between fragments	Can be dragged in, easily added and removed, text changed (both ways) and size can be adjusted according to a handler	Can be dragged in, easily added and removed, text changed (both ways) and size can be adjusted according to a handler	Pass	Size handler only adjustable as soon as the component is selected
TE1.1	A shape with an editable text element can be added	1; Drag some text editable component into the sidebar, like *Action* of *Activity Diagram*.	The component is dragged into the editor	The component is dragged into the editor	Pass	

TE1.2	A shape with an editable text element can be added and edited	1; Repeat TE1.1 4; Doubleclick text in editor to edit it 5; Doubleclick text in sidebar to edit text	The text is editable by way of inputting keys and text updates into both ways (sidepanel <-> inline)	The text is editable by way of inputting keys and text updates into both ways (sidepanel <-> inline)	Pass	
TE2.1	Whenever inline or property panel text is edited the text toolbar pops up	1; Repeat TE1.1 2; Doubleclick text inline	Inline text formatting panel pops up	Inline text formatting panel pops up	Pass	
TE2.2	Special symbol can be added to text	1; Repeat TE2.1 2; Click on symbols menu 3; Click on a symbol	Symbol should appear	Symbol appears	Pass	
TE2.3	Text can be coloured	1; Repeat TE2.1 2; Click on colours menu 3; Click on a colour	Entire text should be selected colour	Entire text is selected colour	Pass	
TE2.4	Text can be underlined, bold and italic	1; Repeat TE2.1 2; Click on bold 3; Click on italic 4; Click on underline	Text should be bold, italic and underline all at the same time	Text is bold, italic and underline all at the same time	Pass	
TE2.4	Text can be underlined, bold and italic with keybinds	1; Repeat TE2.1 2; press CTRL+B (bold) 3; press CTRL+I (italic) 4; press CTRL+U (underline)	Text should be bold, italic and underline all at the same time	Text is bold, italic and underline all at the same time	Pass	
TE2.5	Text can be monospace	1; Repeat TE2.1 2; Click font icon	Text should be monospace	Text is monospace	Pass	
TE2.6	Text size can be changed	1; Repeat TE2.1 2; Click on font+ or font- button	Text font size has changed	Text font size is changed	Pass	
TE2.7	Text alignment can be changed	1; Repeat TE2.1 2; Click on left/right align and swap between	Text align has changed	Text align is changed	Pass	
AI1.1	The page loads two separate editor instances	1; Open webcomponent.html test suite	Two separate editors have been loaded	Two separate editors have been loaded	Pass	
AI1.2	The editor can be toggled to readonly	1; Repeat AI1.1 2; Press *toggle readonly* button	Interaction with objects cannot be done	No interaction with objects	Pass	
AI1.3	Data can be loaded to an editor instance	1; Repeat AI1.1 2; Press load data	Data is loaded	Data is loaded	Pass	
AI1.4	The editor instance can save its data	1; Repeat AI1.1 2; Press getdata button	Data query should match what is in the editor	Data is gotten	Pass	

F AI statement

During the preparation of this work the authors used Microsoft Copilot in order to do quick prototyping, generating boilerplate code, and finding bugs. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the work.

G Maintainer Manual

Maintainer manual

This manual is a PDF-version of the wiki, as seen on the UTML2 wiki page. For more recent updates, the wiki should be consulted. This manual was consulted at 15-04-2026.

Configuring UTML2

UTML2.0 can be configured through some means.

To configure UTML2 you can copy the `.env` in the repository to `.env.local`. `.env.local` overrides the `.env` file and is ignored in git by default.

Configuring predefined diagrams

You can control which diagram palettes are available with the `VITE_DIAGRAMS` environment variable. When `VITE_DIAGRAMS` is an empty JSON array (`[]`), users can choose diagrams from the URL path segments (for example `/class/flowchart`). When `VITE_DIAGRAMS` is a non-empty JSON array, those values are enforced at startup and URL-based diagram selection is ignored. Each value must match a diagram slug (such as `class`, `activity` ...) which is preconfigured in `utils/collection.ts`. This lets you run either a flexible URL-configurable app or a locked preconfigured app from .env.

```
# URL-configurable mode
VITE_DIAGRAMS=[]

# Example URL in configurable mode
http://localhost:5173/class/flowchart

# Preconfigured (locked) mode
VITE_DIAGRAMS=["class","flowchart"]
```

A/B testing

TO A/B-test, an environment variable has been created, called `VITE_USE_ALTERNATE_VERSION`. It can be accessed in Vue by `import.meta.env.VITE_USE_ALTERNATE_VERSION`. Since the site gets compiled to static JS, the .env files get built into the JS, so the A/B configuration is done at build-time. When a new commit is published to the `main` branch, an image with `latest` will appear, but also an image with `alternate`.

How to Parse

This page will tell you how to parse a .utml file!

Get started

The .utml file contains a large JSON object with all the shapes in it. At the root, there is always the `Document`, with the child `Page`. The children of `Page` contain the components which you are parsing.

The first thing which is important, is to understand that what might visually be a single component in the editor, is made up of multiple shapes. Every root of the component has a `proto = true` property, so to find the components you would need to filter on `proto = true`. In addition, the .utml file follows a structure of children and parents, instead of a node-edge structure. That means that everything can be found by recursively traversing the structure.

A practical parsing workflow

Use this order. It is robust and easy to explain to students.

1. Parse JSON and verify the root object is `Document`.
2. Read `Document.children[0]` as the single `Page` used by this app.
3. Recursively traverse that `Page.children` tree.
4. Build a map from `id -> object` while traversing.
5. Resolve cross-references (especially connectors) using that map.

This gives you both hierarchy (parent/children) and cross-links (connector endpoints).

Note: In this app, submissions are expected to contain one page, so you do not need to iterate over multiple pages.

Parent, children, and what counts as a component

A .utml diagram is primarily hierarchical.

- `children`: the nested objects you should recurse into.
- `parent`: usually the parent id for that same relation.

For exam parsing, the most reliable approach is:

- Traverse using `children`.
- Use `parent` as a consistency check.
- Treat objects with `proto = true` as component roots/templates.

Important: some `proto = true` shapes are nested inside other `proto = true` shapes that are `containable`. So do not only inspect `Page.children`; recurse through the full tree when collecting components.

Because one visible symbol may be built from several nested shapes, counting only top-level `proto = true` objects is often closer to what students intended as “components”.

Object identity and classification

These fields are the ones you will use most while grading:

- `id`: unique identifier (used for references).
- `type`: runtime shape class (`Text`, `Ellipse`, `Connector`, `ComputedConnector`, etc.).
- `name`: human-facing label for the object.
- `tags`: semantic labels (`state`, `connectorLabel`, `endState`, ...).

A practical rule:

- Use `tags` for semantic grading criteria.
- Use `type` for structural checks.
- Use `name` for display and reporting.

When students place multiple instances of the same component, those instances will usually share the same `name` and `tags`, but each instance must still have a different `id`.

Connector model (how links actually work)

Connectors are regular objects, not a separate edge table. The type of a connector can be `Connector`, `ComputedConnector`, `Arc` or `ComputedArc`.

Typical connector fields:

- `head`: id of target object at one end.
- `tail`: id of source object at the other end.
- `headAnchor` and `tailAnchor`: normalized anchor coordinates (usually values in `[0,1]`).
- `path`: geometry points for routing.

How to parse them:

1. Read connector objects during traversal.
2. Store unresolved `head`/`tail` ids.
3. After traversal, resolve ids via your `id -> object` map.
4. If an id is missing, keep the endpoint null and flag as invalid link.

Connector labels and label type

Connector labels should be parsed from the connector `labels` array first.

Practical extraction strategy (recommended):

1. For each connector, inspect `labels` first.
2. If `labels` is missing/empty, fall back to connector `children` with `connectorLabel` tags.
3. For fallback parsing, determine position from tags (`begin`, `middle`, `end`) or anchor position (0.1 (begin), 0.5 (middle), 0.9 (end)).
4. Read label text from `simpleText` when available, otherwise fall back to rich `text` parsing.

Label type note:

- Do not classify connector semantics only by `tags`.
- Connector type can change through parsed data, so prefer parsed data over tag inference.
- In this codebase, you may encounter `parsedData.connectorType` (common in computed connectors).

Short rule for grading scripts:

- Primary: `parsedData.connectorType` (if present), else `tags`
- Secondary fallback: infer from tags only when parsed data is missing.

Example of labels

```
"labels": [
    {
        "position": "begin",
        "text": "Begin label",
        "modifiable": true
    },
    {
        "position": "middle",
        "text": "Middle label",
        "modifiable": true
    }
],
```

Text fields

In this app, teachers can use `simpleText` directly.

The `text` field may still contain rich JSON for formatting/editing (Tiptap JSON format), but for grading scripts you should use `simpleText` first.

Practical rule:

- Prefer `simpleText`.
- Only fall back to parsing `text` (Tiptap JSON format) for otherfiles where `simpleText` is missing.

Computed/Configurable components

Some components can be configured in the sidebar, for instance, the Class Diagram. As said before, a component can be multiple shapes. The class diagram is one of those. A seperate shape is the class diagram, title, and a seperate shape is the class diagram box with methods and properties. Take some time to explore the JSON submission of your students, or test a submission.

All configurable components have a property; `configurable = true`.

Every configurable component contains a `parsedData` property with some data which is used to keep this data in a nice format. When importing a save (.utml file), the

program looks at `parsedData` to find out how to render e.g. text. The schema (how the shape of `parsedData` looks) is different per `componentId`, which describes how the shape should act. For instance, a `ComputedText` shape with `componentId` of `classdiagram_class_fields` has the shape below in the example. This is what an export of a class diagram might look like for `classdiagram_class_fields`:

```
{
"type": "ComputedText",
"name": "Class fields",
"componentId": "classdiagram_class_fields",
"parsedData": {
    "fields": [
      {
        "fieldName": "Class field name",
        "fieldVisibility": "Private",
        "fieldType": "Type return text",
        "fieldStatic": true
      },
      {
        "fieldName": "field 2",
        "fieldVisibility": "Protected",
        "fieldType": "Some type",
        "fieldStatic": false
      }
    ],
    "methods": [
      {
        "methodName": "Method 1",
        "methodVisibility": "Public",
        "methodParams": "a,b",
        "returnType": "type",
        "methodStatic": false
      },
      {
        "methodName": "Method 2",
        "methodVisibility": "Public",
        "methodParams": "1234",
        "returnType": "abc",
        "methodStatic": true
      }
    ]
  },
  "configurable": true
}
```

Examples

Example 1: Read `simpleText` from a shape

Input snippet:

```
{
  "id": "state-1",
  "type": "Ellipse",
  "name": "State",
  "simpleText": "s_0",
  "text": {
    "type": "doc",
    "content": [
      {
        "type": "paragraph",
        "content": [{ "type": "text", "text": "s_0" }]
      }
    ]
  }
}
```

Minimal usage (TypeScript):

```
function getStudentLabel(shape: any): string {
  if (typeof shape?.simpleText === "string") {
    return shape.simpleText.trim();
  }
  return "";
}

// Usage:
// const label = getStudentLabel(shape);
```

Example 2: Resolve connector endpoints

Input snippet:

```
{
  "id": "conn-1",
  "type": "ComputedConnector",
  "tail": "state-1",
  "head": "state-2",
  "tailAnchor": [0.5, 0.5],
  "headAnchor": [0.5, 0.5],
  "path": [[0, 0], [50, -50]]
}
```

Resolution idea:

```
const byId = new Map<string, any>();
// ...fill map during traversal

function resolveConnector(connector: any) {
  return {
    ...connector,
    tailObj: connector.tail ? byId.get(connector.tail) ?? null : null,
    headObj: connector.head ? byId.get(connector.head) ?? null : null,
  };
}
```

Example 3: Collect `proto = true` components recursively

```ts
function collectProtoShapes(root: any): any[] {
  const result: any[] = [];

  function walk(node: any) {
    if (!node || typeof node !== "object") return;
    if (node.proto === true) result.push(node);
    for (const child of node.children ?? []) walk(child);
  }

  walk(root);
  return result;
}
```

Use this when grading components, including cases where a containable prototype wraps other prototype shapes.

`enable` and `visible`

These fields are easy to confuse.

- `visible`: whether the shape is rendered.
- `enable`: whether the shape is active/editable in interaction logic.

So an object may exist but be hidden (`visible = false`), or shown but intentionally disabled (`enable = false`).

What to validate in student submissions

A compact checklist that catches most issues:

- Root is `Document`, with exactly one `Page` child.
- Every non-root object appears in exactly one parent `children` array.
- Connector `head`/`tail` references resolve to existing ids.
- Required semantic tags are present for the assignment.
- Component counting is done at `proto = true` roots, not every nested child.
- Text checks use `simpleText` (with fallback only for older files if needed).

Implementation note

If you are writing an automated checker, use a two-pass parser:

- Pass 1: recursive traversal + index building.
- Pass 2: reference resolution + validation rules.

This keeps your grading script readable and reduces false negatives when object order varies in JSON.

Anubis Integration

Overview

UTML2 contains a different build configuration for exporting as a webcomponent. This allows the entire application to be used as a seperate HTML element, inside any HTMl file. It bundles vue and runs on raw JS, so it doesn't require any special setup.

Features

The webcomponent has support for the following features

- Configuring which diagram types are allowed
- Setting initial editor state (setting initial shapes in the editor)
- Updating the editor state (putting shapes in the editor)
- Resetting back to initial editor state
- Providing editor state back to anubis
- Support for dark mode is removed
- File saving/loading is disabled for students
- Multiple `<utml2-app>` elements

How to install

Open the program, run `yarn install`, `yarn build:core` and then run `yarn build:wc`.

Include these files on your page:

- Stylesheet: `./dist/utml2-webcomponent.css`
- Script: `./dist/utml2-webcomponent.js`

An example can be found in webcomponent.html in the repository.

You need to serve the your html for the webcomponent to work.

Main methods

- `initialiseApplication({ allowedDiagrams, readonly, initialData })`
  - Starts/configures the editor.
  - `allowedDiagrams` is denoted by the slugs of the diagram types. e.g. `['class', 'sequence']` for only allowing class- and sequence diagrams.
- `retrieveData()`
  - Returns current diagram JSON.
- `setReadonly(true | false)`
  - Enables or disables editing.
- `loadFromJSON(jsonObject)`
  - Replaces editor content from a JSON object.

Events

- `SendDataToExternal`
  - Fired when certain editor changes happen.
  - Read data from `event.detail.data`.

Embedding the editor in a HTML file

To embed the editor in a HTML file, the following should be done:

```
<!doctype html>
<html lang="en">

<head>
  <!-- Provide the following to enable tailwind scaling options -->
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />

  <!-- Provide the editor style -->
  <link rel="stylesheet" href="./dist/utml2-webcomponent.css" />
  </style>
</head>

<body>
  <h1>UTML Editor</h1>
  <!-- Provide the editor element. To configure it, please read the
documentation further -->
  <utml2-app id="utml1" width="100%" height="80vh"></utml2-app>

  <!-- Provide a script which loads the editor elements -->
  <script type="module" src="./dist/utml2-webcomponent.js"></script>
</body>
</html>
```

Configuration

Configuration can be done by calling a single JS function. the `shapeSet` is a string array which consists of the slugs of all diagram types which the user wants to enable. If the list is empty, then all diagram types are allowed. `diagram` contains the JSON-data that is fed to DGM.js. It can be a JSON-object, or a stringified JSON-object.

```
  <utml2-app id="utml1" width="100%" height="80vh"></utml2-app>
  <script type="module" src="./dist/utml2-webcomponent.js"></script>
  <script type="module">
    // Gets the utml1-app element by its id
    const utml1 = document.getElementById('utml1')

    //Adds handler for when data is saved, JSON is in event.detail.data
    utml1?.addEventListener('sendDataToExternal', (event) => {
      console.log("[2] sendDataToExternal", event.detail.data)
    })

    // Initialize here
    utml1?.initializeApplication({
      shapeSet: ["class"], //CAUTION! Previous UTML used a JSON object, we use
a list of ALLOWED diagram types
      diagram: {...}, //The initial JSON data that is loaded into UTML
      isReadOnly: false,
    })
```

```
  //With the following functions you can interact with utml
  const dataGetButton = document.getElementById("getData")
  dataGetButton.onclick = () => {
    console.log(utml1.retrieveData())
  }

  const readOnlyButton = document.getElementById("toggleReadonly")
  readOnlyButton.onclick = () => {
    utml1.setReadOnly() //Makes it readonly
    //utml1.setReadOnly(false) // also possible to pass in boolean
  }

  const loadDataButton = document.getElementById("loadDataButton")
  loadDataButton.onclick = () => {
    utml1.loadData({...}) //Insert JSON of UTML file here
  }

</script>
```

Headless Browser

About

The headless browser is a command-line tool written in Python to mass-convert many `.utml` files to `.svg` files. It uses Selenium and the script is located in the `browser` folder.

How to run

1. Run `python -m venv .venv` to create a virtual environment folder
2. Activate it (`source .venv/bin/activate`).
3. Install selenium
4. `pip install -r requirements.txt`
5. Install selenium driver (we're using the chromium driver)
6. View usage and arguments: `python main.py -h`
7. Run the converter:
8. Single file: `python main.py <url> <path-to-file.utml>`
9. Whole folder: `python main.py <url> <path-to-utml-folder>`
10. Custom output folder: `python main.py <url> <input> -o <output-folder>`

Notes:

- Exported SVG files are written to `output/` by default.
- Use `-o` / `--output` to choose a different output directory.

6. When done, exit the environment by writing `deactivate`

DGM.js

DGM.js is an external library maintained on GitHub. Its goal is to provide an editor with much flexibility, it offers features such as pages, smart-shapes that can have constraints or scripts, and many kinds of interactions with the editor.

Too simplify the process of understanding DGM.js we will provide a small overview of the innerworkings. DGM.js also has some documentation First, the DGM.js project consists out of multiple modules. The main project simply is a simple react application to demonstrate dgm.js and these submodules.

DGM.js Core

The most important code for DGM.js can be found in the core package within the repository.

Within this package, most file names are clear. The most important file is `editor.ts`. The editor combines a lot of code into a functioning editor canvas. It keeps track of state, changes and offers many useful functions to interact with the canvas or state.

Shapes are the components that will be rendered on the screen. These are defined in `shapes.ts`, not all shapes are actively used within DGM and are

`macro.ts` contains all util functions in regards to transactions, for instance moving shapes.

Within the editor there are two ways of interacting with canvas, either with a Handler or with a Manipulator.

Handlers

Handler is the internal name for what would be a tool in the editor. For instance, there is a selection handler, which handles the logic for when you have the selection tool selected. There is also one for making rectangles, ellipses, etc. Simply look inside the Handler directory to see what is already supported.

Within `editor.ts` the list of handlers is stored.

Manipulators & Controllers

Manipulators provide logic to interact with a specific shape, such as moving, rotating, and changing the size of a box. Inside of the Manipulator, Controllers are initialized to handle each specific interaction, the Manipulator is simply to bind this to the given Shape type. The manipulators are registered inside of the `manipulatorManager` inside the editor, it is simply a record from Shape type to the manipulator it should use.

By adding or changing Controllers, you can change the behaviour inside the editor. For instance, for UTML we wanted to changed the snapping from inside a rectangle to the edges of a rectangle. We simply changed the code inside `update` for the `ConnectorReconnectController`. I advise you to take a look at a few Controllers, the overall structure is simple.

Constraints

Constraints are simple pieces of logic which can be applied to shapes. They have a `zod` schema to take settings. Most constraints are used to make shapes or sets of shapes behave dynamically. For instance, there is a constraint for a shape to align to its parent.

Within DGM.js, it tries to meet constraints within 3 iterations after a change has been applied. This happens in `macro.ts` in the functions `resolveShapeConstraints` and `resolveAllConstraints`.

The constraints are registered in `shapes.ts` in the constraintManager.

Transactions

Whenever anything changes in the editor, this is sent and stored within a `Transaction`. DGM.js supports undo/redo and keeps track of a history internally. Transactions are the core of this, as they store what they change was and which shapes were affected. Please look into `macro.ts` and `actions.ts`.

Adding shapes programmatically

To add shapes to a page, use `ShapeFactory` within `factory.ts`. This can be accessed as follows

```
// create a rectangle
const rectangle = editor.factory.createRectangle([[0, 0], [100, 100]]);
rectangle.fillColor = "#ff0000";
rectangle.strokeColor = "#00ff00";
rectangle.strokeWidth = 3;
rectangle.roughness = 2;

// insert into current page
editor.actions.insert(rectangle);
```

Mal

Mal is the scripting language of DGM.js which can be inserted into a shape to dynamically update.

dgmjs-plugins-yjs

This is the package which includes the collaboration plugin for dgm.js

export

This is a package which provides crucial util functions for exporting the canvas as any non-json and non-pdf format.

pdf

This is a module that exports the canvas pages as a pdf file.

react

This is a wrapper which takes the core editor, and turns it into something that works well with react. We do the same inside UTML. It also provides some other Components that handle shape previews, text editing toolbar and text editing.

DGM Component Structure

DGM.js uses a JSON structure to keep track of the objects within the editor. The general relation between objects is tracked by specifying a `parent` and `children`. For the parent only an id is passed, however for children the complete json of the child is passed into the list.

It is important to understand that DGM.js does not use Nodes and Connections. Rather it uses generic classes and extends upon those to provide wanted functionality

```
Obj -> Shape -> ...
```

Each object in the editor inherits Obj, this is the base class for every element that exists within the editor. it has an UUID, parent, children, serialization, and traversion.

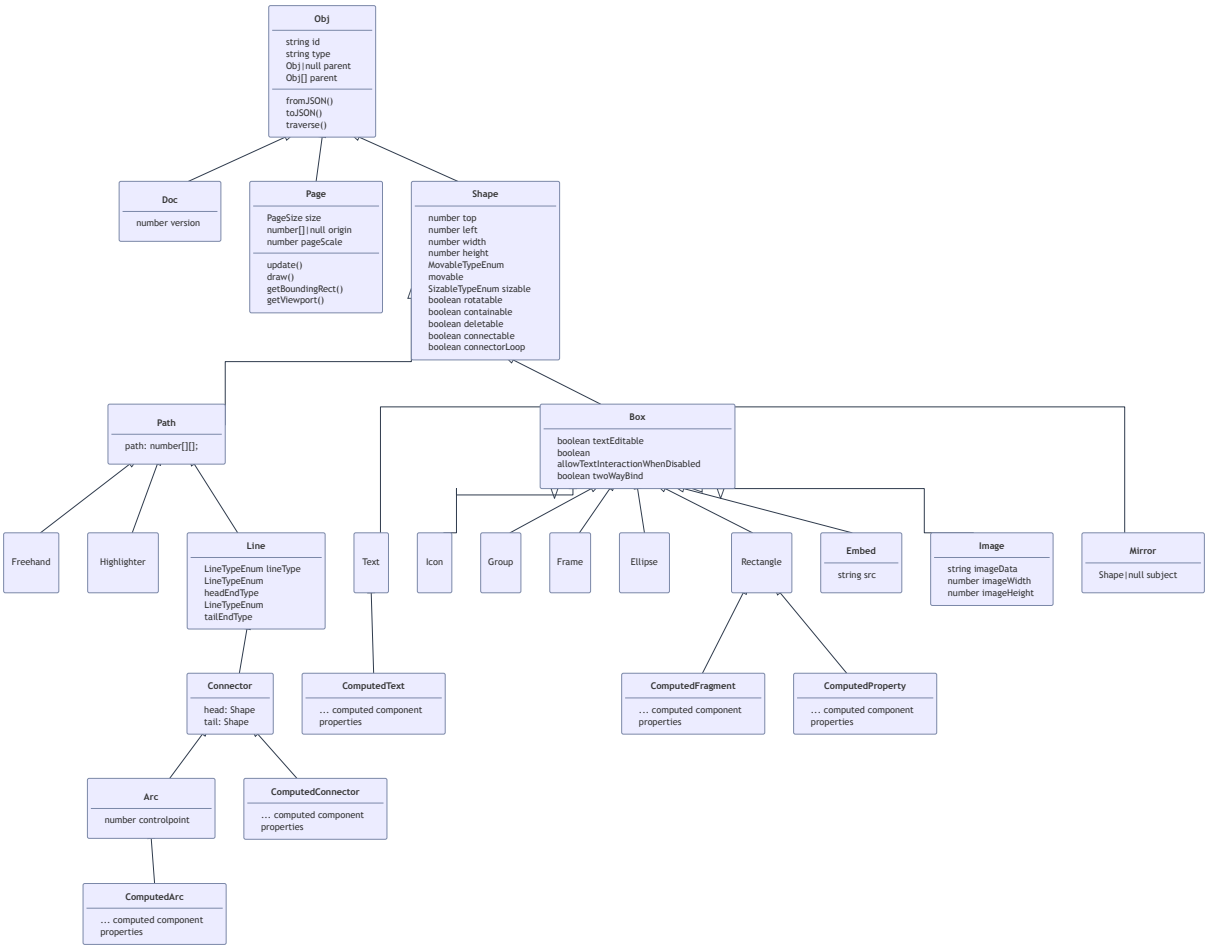
The Shape class expends upon Obj, by taking into account much more such as coordinates. It is the implementation of Smart Shapes within dgm.js.

The cases where a class only extends Obj and not Shape is very limited. For instance, within the editor, there is first a Document class, which has a Page class as a child. A page then has Shapes as children. Document and Page use Obj, since they don’t have coordinates and cannot be interacted with directly.

Most other classes extend Shape, for instance, Rectangle, Ellipse, Text, etc.

Newly added Shapes for UTML

- Arc
- SnapPoint
- Computed Components


Obj
string id
string type
Obj|null parent
Obj[] parent
fromJSON()
toJSON()
traverse()
Doc
number version
Page
PageSize size
number[]|null origin
number pageScale
update()
draw()
getBoundingRect()
getViewport()
Shape
number top
number left
number width
number height
MovableTypeEnum
movable
SizableTypeEnum sizable
boolean rotatable
boolean containable
boolean deletable
boolean connectable
boolean connectorLoop
Path
path: number[][];
Box
boolean textEditable
boolean
allowTextInteractionWhenDisabled
boolean twoWayBind
Freehand
Highlighter
Line
LineTypeEnum lineType
LineTypeEnum
headEndType
LineTypeEnum
tailEndType
Text
Icon
Group
Frame
Ellipse
Rectangle
Embed
string src
Image
string imageData
number imageWidth
number imageHeight
Mirror
Shape|null subject
Connector
head: Shape
tail: Shape
ComputedText
... computed component properties
ComputedFragment
... computed component properties
ComputedProperty
... computed component properties
Arc
number controlpoint
ComputedConnector
... computed component properties
ComputedArc
... computed component properties

Queries

Queries

You can use query expression to filter shapes. This is sometimes used in Constraints. The syntax of query expression can be defined as below.

```
<query>         = <clause>["|" <clause>]*
<clause>        = <term>["&" <term>]*
<term>          = <name-selector> | <type-selector> | <tag-selector>
<name-selector> = <name>        e.g.) OuterBox, TextName, ...
<type-selector> = "@"<type>     e.g.) @Box, @Text, @Line, ...
<tag-selector>  = "#"<tag>      e.g.) #label, #compartment, ...
```

Here are some examples of query expressions:

- Foo : All shapes whose name is Foo.
- @Text : All shapes whose type is Text.
- #compartment : All shapes having a compartment tag.
- Bar&@Line : All shapes whose name is Bar and type is Line.
- @Box|Baz|@Text&#compartment All shapes whose type is Box or name is Baz or type is Text with a compartment tag.

Constraints

Constraints

It is possible to add constraints to shapes, such that they can dynamically respond to other shapes. For instance, you could have a box with text, where the text has a constraint such that it always aligns to the middle of the box.

To add constraints to a shape you can add the following besides the other properties the shape has:

```
constraints: [
  {
    id: "scaling-anchor-to-parent",
    horAnchor: 0.1,
    vertAnchor: 0.9,
  },
  ... (more constraint objects)
]
```

Every constraint has an id, but the other information differs between the types of constraints.

All existing constraints can be viewed in here

Custom Added Constraints

scaling-anchor-to-parent

This constraint anchors the shape to its parent at a given anchor point. Uses the shapes top-left point.

options

- **horAnchor**: float number from 0 to 1, where 0 is completely to the left of the shape, and 1 completely to the right of the shape
- **vertAnchor**: float number from 0 to 1, where 0 is the top of the shape and 1 the bottom.

Shape Properties

Shape Properties

Based on the shape, you can set a lot of properties. It depends on what classes it extends which are available to you.

To check the base properties available in DGM.js, click here.

Added properties

- connectorLoop, the head and tail of a connector cannot be connected to the same shape. By enabling this it is allowed.
- connectableOptions, a string list with the allowed connection points on a shape. can be: ‘bounding-corners’, ‘bounding-midpoints’, ‘middle’, ‘outline’. Read more in Snapping
- twoWayBind, explicitly enable text editing in the sidebar (is often also done implicitly). Needs to be turned on when a Shape is disabled and you want text editing in the sidebar.
- allowTextInteractionWhenDisabled, allows you to inline edit text, even though this shape is disabled for interaction.
- componentId, explained in Computed Component
- parsedData, explained in Computed Component
- deletable, by default all shapes are deletable. To disable deletion, you can set this to false.

Snapping

Snapping Configuration

Connector snapping is configured per shape through these JSON properties:

- `connectable`: whether the shape accepts connector ends
- `connectableOptions`: which anchor strategies are available

Supported `connectableOptions` values:

- `"bounding-corners"`
- `"bounding-midpoints"`
- `"outline"`
- `"middle"`

Example

```
{
  "type": "Rectangle",
  "connectable": true,
  "connectableOptions": [
    "bounding-corners",
    "bounding-midpoints",
    "outline"
  ]
}
```

Examples of simpler configurations:

- disable snapping: `"connectable": false`
- corners only: `"connectableOptions": ["bounding-corners"]`
- outline only: `"connectableOptions": ["outline"]`
- center only: `"connectableOptions": ["middle"]`

Priority

If multiple options are enabled, snapping prefers them in this order:

1. `bounding-corners`
2. `bounding-midpoints`
3. `outline`
4. `middle`

Defaults

If no custom `connectableOptions` value is provided, the built-in defaults are used:

- most shapes: `["bounding-corners", "outline", "bounding-midpoints"]`
- `Path`: `["outline"]`
- `Ellipse`: `["outline", "middle"]`

Custom SnapPoints

SnapPoint shapes are separate from `connectableOptions`.

Use them when you want explicit hand-placed connector targets instead of relying only on corners, midpoints, outline, or center snapping. Also use them if you need to parse whether a connector is attached to a specific snappoint.

Arc Shape

Overview

An Arc is a shape that extends connector but only has three points; the head, tail and controlpoint. The controlpoint is relative to the middle of the Arc. This way the control point moves when the head or tail of the shape are moved. It also uses a simple parabolic calculation for its path.

Example structure

It is mostly the same structure as a connector. However, the path contains only the head and the tail. The `controlPoint` is a separate field because this is stored relative to the middle.


```
{
    "id": "USIYUDFBhd72JD6g2ds80ndow63",
    "type": "ComputedArc",
    "parent": "",
    "children": [],
    "name": "Arc",
    "description": "",
    "proto": true,
    "tags": ["automata_arc"],
    "enable": true,
    "visible": true,
    "movable": "free",
    "sizable": "free",
    "rotatable": false,
    "containable": false,
    "containableFilter": "",
    "connectable": true,
    "allowedLabelPositions": ["middle"],
    "controlPoint": [
        0,
        -50
    ],
    "constraints": [
        {
            "id": "adjust-route"
        }
    ],
    "properties": [],
    "scripts": [],
    "pathEditable": true,
    "path": [
        [
            0,
            0
        ],
        [
            100,
```

```
                0
            ]
        ],
        "headEndType": "solid-arrow",
        "headAnchor": [
            0.5,
            0.5
        ],
        "tailAnchor": [
            0.5,
            0.5
        ]
    }
```

Connector Text Labels

Connector text labels in JSON

Text labels are only supported on shapes with the type “ComputedConnector”. If a connector is not a ComputedConnector, the label JSON fields will be ignored.

allowedLabelPositions

Use allowedLabelPositions to enable editable label slots on the connector path.

Supported positions are:

- “begin”
- “middle”
- “end”

If you include a position here, that label position becomes available on the connector.

labels

Use labels to define predefined labels on the connector.

Each label entry supports:

- position: where the label is attached, usually “begin”, “middle”, or “end”
- text: the visible label text
- modifiable: whether the user can change the text in the properties UI

If modifiable is false, the label is treated as fixed/default text. If modifiable is true, the user can edit it.

Example

```
{
  "type": "ComputedConnector",
  "name": "Inheritance",
  "allowedLabelPositions": ["middle"],
  "labels": [
    {
      "position": "begin",
      "text": "hey",
      "modifiable": true
    }
  ]
}
```

Notes

- allowedLabelPositions controls which connector label positions are available.
- labels defines default labels that should already exist on the connector.
- You can combine both: predefined labels in labels, plus extra editable positions in allowedLabelPositions.

Creating a New (Computed) Component Type

Creating components

To create a new component, add a JSON object in one of the collections, exposed by `/src/utils/collection.ts`. These objects all have some things in common, like:

- `id`
- `name`
- `type`
- `parent`
- `proto`
- … Most of these are described in the DGM component structure When a JSON object is loaded into the canvas, DGM.js looks at the `type` property and creates a class from the instantiator registry. Then, `fromJSON` is called to load data into that class.

Built-in computed runtime classes (`ComputedText`, `ComputedConnector`, `ComputedArc`, `ComputedProperty`, `Fragment`) are registered in core at `packages/dgmjs-utml/packages/core/src/computed/register-builtins.ts` and wired during editor construction in `packages/dgmjs-utml/packages/core/src/editor.ts`.

To create components which can be easily adjusted at runtime, we created `configurableComponent`, which can take a zod schema to generate a form and to render a custom component state based on the input of the form. Adding a non-computed component is quite easy, but the default components already cover enough situations to not need to create new ones.

Creating a New Computed Component Type

Every computed component needs to be bound to a `componentId`, since there are multiple components using e.g. the `ComputedText` runtime type.

There are two concepts to keep separate:

- `componentId`: the configuration entry. Examples: `classdiagram_class_name`, `classdiagram_class_fields`, `arrow_classdiagram`.
- `type`: the runtime class. Examples: `ComputedText`, `ComputedConnector`, and `ComputedProperty`.

Most changes in this codebase are new `componentId` entries for an existing component type. A new computed component type is only needed when you need a new component type class.

Runtime Flow of Computed Components

1. A collection JSON file in `src/collections` defines prototype shapes with `proto: true`. `proto: true` must be enabled for template components.
2. The `type` gets read and the JSON gets converted into a class.
3. `src/utils/collection.ts` exposes those prototype JSON objects as palette items.

4. Dragging a component item into the canvas clones the prototype JSON and deserializes it into a shape instance (`src/utils/useComponentDragDrop.ts`).
5. During `fromJSON`, the runtime class optionally resolves `componentId` through the computed component registry.
6. The property panel builds a form from the Zod schema, validates input, stores it in `parsedData`, and reruns the component's compute logic.

The Main Pieces

Collections

Collections are the JSON documents in `src/collections`, such as `classdiagram.json`.

They do two things:

- define what appears in the palette
- define the default JSON structure of inserted shapes

`src/utils/collection.ts` imports these JSON files, finds all objects with `proto: true`, and exposes them as palette items. A prototype is treated as configurable when it or one of its children has `configurable: true`.

Plugins

The app bootstrap calls `registerEditorComputedFactories()` from `src/utils/registerEditorComputedFactories.ts` via `src/plugins/installUTMLAppPlugins.ts`.

This registers computed component configs, such as:

- computed component configs like `arrow_classdiagram`

Runtime constructors for built-in computed types are already registered by core (`registerBuiltinComputed(...)`) when the editor is created.

If a new runtime type is not registered in the instantiator, collection JSON can reference it but the editor cannot instantiate it.

configurableComponent.ts

`src/utils/configurableComponent.ts` re-exports the registry layer from `@dgmjs/core`.

The implementation lives in `packages/dgmjs-utml/packages/core/src/computed/configurable-component.ts`.

It defines:

- `ComputedComponentInterface`
- `ConfigurableComponentConfig`
- `registerComputedComponent(...)`
- `registerComputedComponents(...)`
- `getComputedComponent(...)`

The important idea is that `componentId` resolves to a config object containing a Zod `schema` and a `computeMethod`.

configurableComponentMixin.ts

`src/utils/configurableComponentMixin.ts` re-exports the shared mixin from `@dgmjs/core`.

The implementation lives in `packages/dgmjs-utml/packages/core/src/computed/configurable-component-mixin.ts`.

The mixin instance carries this shape-specific state:

```
{
  schema?: z.ZodObject<any>
  settings?: ConfigurableComponentConfig<any>
  componentId?: string
  parsedData?: any
  computedSettings?: any
  configurable: true
}
```

What each field means:

- `componentId`: link back to the registered config entry
- `settings`: the registered config resolved from `componentId`
- `schema`: the Zod schema taken from `settings`
- `parsedData`: validated user input that should be persisted in JSON
- `computedSettings`: result of `computeMethod(parsedData)`
- `configurable`: marker used by the UI to show the schema form

The mixin also implements the standard lifecycle:

- constructor: sets `this.type`
- `toJSON()`: writes `type`, `componentId`, `parsedData`, and `configurable`
- `fromJSON()`: restores `componentId`, `parsedData`, and `schema`
- `parseAndSetProperties()`: validates input, stores `parsedData`, and computes `computedSettings`
- `afterSetProperties()`: defaults to `editor.update()` and `editor.repaint()`

If you create a new computed runtime type, start with this mixin rather than reimplementing the lifecycle manually.

JSON Structure

The collection JSON is part of the type system. It is not just static drawing data.

Top-level computed prototype

Example from `classdiagram.json`:

```
{
  "id": "Amk0735xh7DuyxoiK4p8D2",
  "type": "ComputedConnector",
```

```
  "name": "Association",
  "proto": true,
  "componentId": "arrow_classdiagram",
  "configurable": true,
  "default": "Association"
}
```

Important fields:

- `type`: runtime class name, registered with the editor instantiator
- `componentId`: lookup key for schema plus compute logic
- `proto`: whether the object shows up in the palette
- `configurable`: whether the property panel treats it as configurable
- `default`: optional seed value used by the runtime class

Nested computed children

Collection JSON can also contain a regular shape with computed children:

```
{
  "name": "Class",
  "proto": true,
  "type": "Rectangle",
  "children": [
    {
      "name": "Class fields",
      "type": "ComputedText",
      "componentId": "classdiagram_class_fields"
    },
    {
      "name": "Class name",
      "type": "ComputedText",
      "componentId": "classdiagram_class_name"
    }
  ]
}
```

This is why the property panel traverses children as well: the configurable object may be nested inside a larger prototype.

What the mixin writes to JSON

The mixin persists these fields on the shape JSON:

```
{
  "type": "ComputedText",
  "componentId": "classdiagram_class_name",
  "parsedData": {
    "className": "Customer"
  },
  "configurable": true
}
```

That is the minimum structure needed for the runtime class to restore its schema-backed state when a document is loaded again.

Adding a New Component ID to an Existing Type

If you only need a new variant of `ComputedText` or `ComputedConnector`, you do not need a new runtime class.

Example for `ComputedText`:

```
const myComponentConfig = {
  componentId: 'my_component_id',
  componentType: 'ComputedText',
  schema: z.object({
    title: z.string().describe('Title')
  }),
  computeMethod: (data) => `Title: ${data.title}`
}
```

Then include it in a factory list that is registered via `registerEditorComputedFactories()`, and add a prototype to a collection JSON file:

```
{
  "type": "ComputedText",
  "componentId": "my_component_id",
  "proto": true,
  "configurable": true,
  "name": "My Computed Text"
}
```

Creating a New Runtime Type

1. Create a runtime class using `ComputedComponentMixin(...)` with the DGM base class you want.
2. Create a factory module that exports config entries with `componentId`, `schema`, and `computeMethod`.
3. Register the config list in `src/utils/registerEditorComputedFactories.ts`.
4. Register the runtime constructor in the editor instantiator (for built-ins this is in core at `packages/dgmjs-utml/packages/core/src/computed/register-builtins.ts`; external runtime types can be registered through the computed registry API in core).
5. Add one or more `proto: true` entries in a collection JSON file using the new `type`.
6. If needed, override `parseAndSetProperties(...)` or `afterSetProperties(...)` for extra derived behavior.

`ComputedConnector` is the main reference for type-specific behavior because it does more than the default mixin: it seeds defaults from JSON and applies computed results onto connector fields and label children.

Quick Checklist

1. Pick whether this is just a new `componentId` or a whole new runtime `type`.

2. Define a Zod schema and `computeMethod`.
3. Register the config.
4. Register the runtime constructor if the `type` is new.
5. Add a collection prototype with `proto`, `type`, `componentId`, and `configurable`.
6. Confirm the computed result is actually applied by the runtime class.

Reference Files

- `src/utils/configurableComponent.ts`
- `src/utils/configurableComponentMixin.ts`
- `src/utils/registerEditorComputedFactories.ts`
- `src/plugins/installUTMLAppPlugins.ts`
- `src/utils/computedText/computedText.ts`
- `src/utils/computedText/computedTextComponentFactory.ts`
- `src/utils/computedConnector/computedConnector.ts`
- `src/utils/computedConnector/computedConnectorComponentFactory.ts`
- `src/utils/computedProperty/computedProperty.ts`
- `src/utils/computedProperty/computedPropertyComponentFactory.ts`
- `packages/dgmjs-utml/packages/core/src/computed/configurable-component.ts`
- `packages/dgmjs-utml/packages/core/src/computed/configurable-component-mixin.ts`
- `packages/dgmjs-utml/packages/core/src/computed/register-builtins.ts`
- `src/utils/collection.ts`
- `src/utils/useComponentDragDrop.ts`
- `src/components/ComponentProperties.vue`
- `src/collections/classdiagram.json`

Custom SnapPoint

Custom SnapPoints

`SnapPoint` shapes are separate from `connectableOptions`.

Use them when you want explicit hand-placed connector targets instead of relying only on corners, midpoints, outline, or center snapping. Also use them if you need to parse whether a connector is attached to a specific snappoint.

To use Custom SnapPoints you can add the following to the children of the shape where you want to add SnapPoints

```
{
  "id": "test2",
  "type": "SnapPoint",
  "parent": "M50a-IhLQoMYPmo3Z9EJ5",
  "name": "Cool Snappoint 2",
  "description": "input for instance",
  "constraints": [
    {
      "vertAnchor": 0.75,
      "horAnchor": 0.5,
      "id": "scaling-anchor-to-parent"
    }
  ],
  "tags": [
    "snappoint-input2"
  ],
  "fillColor": "$red9"
}
```

The position of the snappoint is determined by the scaling-anchor-to-parent constraint. VertAnchor is the vertical position relative to the parent, 0 is the top, 1 is the bottom. HorAnchor is the same, 0 is the left, 1 the right. It is also possible to not use the constraint, if you do not want the point to move with the shape if it is resized.

To change the color of a SnapPoint, change `fillColor` to the desired color.

Two-way Binding

UTML2 supports two-way binding for editing text in both the sidebar and in the editor. This is done by the property panel listening for selection, only if a certain criteria is met;

```
object.type != 'Computedtext' && object.visible &&
        (object.twoWayBind || (object.textEditable && object.enable))
```

So, if an object is editable (by definition ComputedText is not editable), visible and enabled it should be two-way-bound. However, in some cases an object which is not enabled (one is not allowed to edit the object which contains the text) needs to be two-way bound. For this, the property `twoWayBind = true` can be set on an object to force the property panel to bind to it. There is a special case, namely `Connector`s or `Arc`s. When one of those is selected, the system shows a special interface for editing and creating connector labels.