Universiteit Leiden
The Netherlands

Opleiding Informatica

Enabling collaboration in a model design environment

Sven Hepkema (s2454556)

Supervisor: Guus Ramackers
Second Supervisor: Joost Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

August 22, 2023

Abstract

This thesis devises an approach to enable multi user concurrent collaboration in an existing model design environment. This existing environment is ngUML, a UML based modelling tool, which currently does not support multiple users. To let users collaborate in an existing environment, a tool is designed to let users share work, as well as a new protocol to synchronize diagrams when multiple users are working on it. The design and implementation takes into account that it needs to be easy to integrate with the existing code, easy to understand for future developers and easy to extend for future functionality. Tests were executed to measure the increased load on the server for the synchronization protocol and users were interviewed to receive their feedback on the new functionality.

Contents

1 Introduction

1.1 Problem statement

This thesis will research how to enable end users to collaborate concurrently while using the software built by the ngUML project. The ngUML project built software that generates applications from natural language text. The applications are data focused, and describable by a UML diagram. The natural text is converted into a UML diagram, which is viewable and editable by the user. The user can then let the program generate the application from the UML diagram and use it in the browser.

For example, the user describes three objects, a consumer, a product and a vendor. The user describes the relationships between the objects, the consumer can buy one or multiple products from the vendor, and the vendor receives payment from the customer. The user also defines a property on the product and the consumer, the product has a name, and the consumer has an address. The UML diagram is then generated and reflects that, it shows three classes with the properties name and address attached to product and consumer respectively. There are lines between them, which represent the previously described relationships between the classes. The user might make some adjustments to the properties, he for example forgot to also give the product a price. Then the user decides he is satisfied with the UML diagram, and tells the program to generate the application. He can now actually use his example, which he constructed with natural language.

It would enhance the potential of the software if users could collaborate while writing the prompt, or while adjusting the UML diagram. Currently there is also another problem; the implementation of the program only allows for a single instance of an application to be constructed. It is not possible to work on multiple applications, without deleting the previous one. This severely limits the usability of the software for end users.

1.2 Solution direction

The focus of the thesis is to allow teams to collaborate concurrently on designing diagrams via the ngUML software. The design of the concurrent API architecture should be generic enough to easily use it for implementing concurrent collaboration in any part of the project in the future. To further enable collaboration, users should have a method to share their work with eachother, across diagrams.

This means that the redesign of the api will need to choose an approach to share data in real time. That approach should then be easily portable for any new functionality as well. Introducing collaboration also requires a way for users to register themselves and for them to view their projects. Users will need to collaborate within groups of users. These groups must be able to share diagrams or partial diagrams with other teams to be able to collaborate within for example a larger company or class. The software produced during this thesis will also need to implement this, and will also give recommendations for future functionality, such as a more in depth user rights system or for example version control. I will also need to design in what way users will collaborate. A git like system could be used, in which a user can decide when to add his changes to the main branch while also enabling users to test their ideas in a separate branch. Another option could be a system where the user can view real time changes to the diagram. The answer to these design questions will also play a big role in which design pattern is most suitable to implement concurrency.

1.3 Research objectives

The thesis will produce the following items to implement collaboration in the ngUML project. Adjustments in the existing code base will be needed to transform it from a single user to a multi user system. It focuses on creating an extensible design, that is both very intuitive for users, as well as simple for developers to work with. It should be easily extensible, as future developers may then build upon the foundations laid by this thesis. It should refrain from introducing any elements that cannot easily be understood or require extra learning by new developers, such as external libraries or complex protocols. A method will also be designed to let users share their work. At the end, the implementation of both the concurrent collaboration system and the system to share work should be tested quantitatively and qualitatively.

- Designing a new, generic approach to let the existing API support concurrent collaboration by users.
- Implementation of concurrent collaboration in the UML design stage in the UI.
- Designing a method to let users share their work with others, across diagrams.
- Evaluation of the concurrency both quantitatively (stress tests) and qualitatively (real user test with questionnaire to evaluate the quality of collaboration).
- Recommendations for future collaboration functionality and tools.

1.4 Thesis overview

This thesis starts off with a section discussing work by others related to this thesis and it's design goals. In this section the viability of three different approaches for the real time data transfers are researched. In the architectural design section the target audience is discussed and an overview of the design is given. After that overview, various aspects of the design will be discussed in detail, each time providing several options, and then a section which discusses what is the best option and why. In the implementation section the actual implementation of the design is discussed. In this section there will be screenshots showing the UI that was implemented, as well as code snippets that show how the most important concepts are translated into code.

2 Related Work

2.1 UML

UML [Obj17] is a language to model complex systems. The language does not use symbols, but very precise rules around schematic specifications. These rules allow designers to make intricate models of their ideas before implementing them, or to map an existing system into a clear overview. The majority of developers use UML for their modelling needs [FL08]. They usually do not follow the strict rules of UML, but use a more flexible form of it. There is a large variety of existing UML modelling tools, of which the majority allows various ways to let users collaborate [Ozk19] This illustrates the need to enable collaboration in modern UML tools.

2.2 Real time data transfer

Collaboration requires seeing the changes that another team member makes while you are working together. To support this, it is possible to use either a polling or long polling based approach [LSASW11] to support this via the http protocol [BLFF96]. A polling approach is the most rudimentary solution, requiring no external libraries to implement. As it is quite simple, it is also quite easy to understand. An alternative to http requests is the use of websockets [FM11]. Websockets were designed to support real time, bidirectional communication. The amount of overhead is reduced, and it requires only a single TCP connection. It is designed to be able to be used in environments with http requests.

In a study by Pimental & Nickerson [PN12], they studied several connectivity metrics for three connection models for a single application. The three models were polling, long polling and websockets. The application used a low volume (4 Hz, or 4 requests per second) of continuous real time requests. The study found that long polling performed very similar to websockets, even on longer distances (Canada-Japan). On shorter distances, polling was able to sustain a 4 requests per second and only had an average latency three times larger than websockets. This might seem large, but for applications which do not require low latency, high volume communication, this could be a very acceptable performance for a simple protocol. Django can be used to implement polling, as it does not require any unusual http requests. The Django Channels library [DJC] enables Django to support both long polling as well as websockets.

2.3 Load testing

To verify that the implementation is adequate for the use case, The limits of the application are tested by *load testing* it. Load testing evaluates the behaviour of software under high pressure [PS19], in this case large numbers of users. There are two important numbers that will need to be measured after the design is implemented. The first statistic is how much of an impact the new concurrency proof API has on the server load. The second metric is how many users can work concurrently on a single server. This will be important as it indirectly gives us a limit on the maximum amount of users. If this user limit proves problematic in the future, the limit might be improved by revising the server architecture. The new architecture design can then be benchmarked by using the same stress test to ensure that the new design improved the maximum amount of concurrent users.

To test the maximum load capacity of the servers, Requests are generated by a script to simulate a user using the system. It is important that this workload closely resembles either real scenario's, or stresses the system in such a way that you can measure your target indicators on different loads. Especially session-based interaction is highly susceptible to dependencies between requests [WTSW95]. This means that a simple load test consisting of an arbitrary set of requests fired off at uniform intervals does not adequately resemble a real workload. You should closely model real workloads, and then use appropriate tools to mimic a real world environment [KRM06]. Often, load testing results might be inflated because the load is only tested for a small amount of time. *Soak testing* is used to test the performance of the server when it has to endure high loads for longer periods of time [HC15].

To generate the requests that the server will receive it is possible to use a script. There are multiple options for this, for example writing a custom python script or tools like Locust [HBHH], which is an open-source scriptable testing tool for load testing. It allows the user to specify workloads, and then simulate any number of users. The main requirement is that the workload accurately represents what you would like to measure, and that you can generate enough requests to test the server load.

2.4 Technology acceptance model

To evaluate the user adoption of the new functionality, the *Technology Acceptance Model* (TAM) is used. This model was devised by Fred Davis, who had the following goal in mind:

> "The goal of this research is to develop and test a theoretical model of the effect of system characteristics on user acceptance of computer-based information systems."

He wanted to know which variables influence user acceptance, and use that understanding to create a systematic and practical method to evaluate prototype information systems. He identified two determinants that indicate whether or not a user would like to use the system. The first determinant is *perceived usefulness*, indicating to what extent the user thinks the system will help him. The other determinant is *perceived ease of use*, indicating to how difficult the user thinks it is to use the system. Keyword here is *perceived*, as it measures the perception of the user, and not the real effectiveness or adoption time of the system. In figure 2.4, the causal relationships between the two determinants are shown. Davis hypothesized that both determinants are significant for the attitude towards use, but do not directly affect system use. In other words, the determinants do not have a direct effect on system use, but they do have an indirect effect when you control for attitude towards system use. [Dav85]

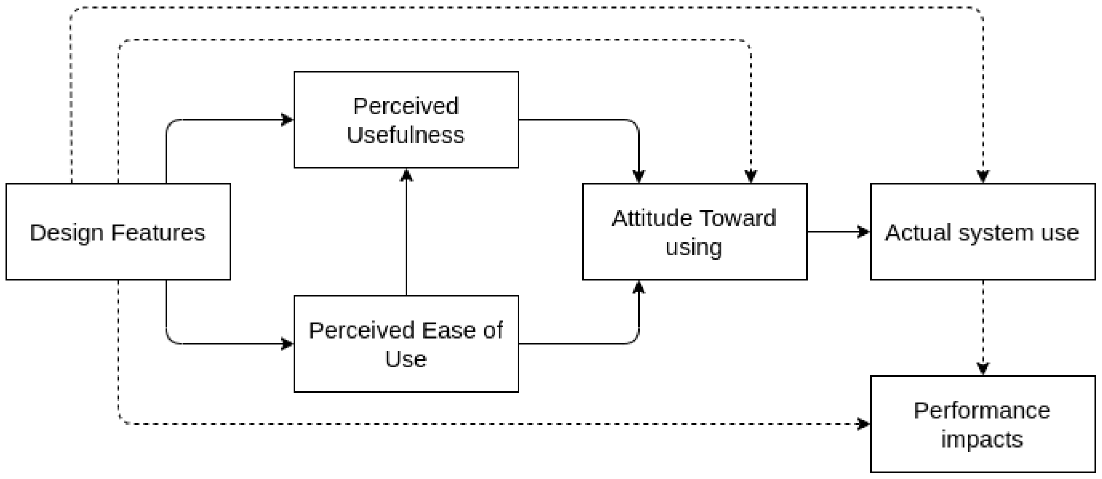


Figure 1: This figure shows the causal relationships between the design features and the performance impacts for users, based on the system adoption. The dotted lines indicate causal relationships that are not studies with the TAM, and the other lines show the relationships studied by TAM. This figure was copied from [Dav85]

These determinants can then be measured with a questionnaire. The questionnaire designed by Davis asks the user questions such as "I would find using this system to be enjoyable". The user may reply by crossing a box in a range of seven boxes. Each box indicates the user's opinion between a range of two opposing adverbs, such as 'low' and 'high', or 'likely' and 'unlikely'. Questions may not only evaluate the user's experience, but also other aspects, such as the importance of viewing information in a certain way, such as 'in my job, numeric charts are', with the adverbs 'unimportant' and 'important'. Questions like these illustrate how TAM can not only be used to evaluate existing prototypes, but also to identify important features or requirements for a possible new information system. Each section of questions is then concluded with a final question asking the respondents how confident they are in their answer.

TAM has developed itself since 1984 continually [MG15], mostly focusing on uncovering new variables that might play a role in system adaptation. Most studies that use TAM in their analysis suffer from the same aspects. In most studies the user is only questioned once, while their opinion could change over time. The measurement is usually done while the user is still a beginner, and not after sufficient time to get familiar with the system. Some studies suffer from self selection bias by not questioning each user, but instead relying on voluntary participation. Researchers often that a benefit of TAM is that it has standardized the evaluation methods of user satisfaction. A frequent complaint though was that TAM might oversimplify the evaluation. [LKL03]

About half of the studies use students. Younghwa, Kenneth & Kai [LKL03] state that this might hurt the potential for generalizing the results of studies to the general user base. King & Jun [KH06] found however that students can be used as an approximation, or surrogate for professional users. They often displayed similar behaviour, while extrapolating students' behaviour to that of general users should be avoided. Another important finding by King & Jun was that while perceived usefulness was more important than perceived ease of use for professional products, this did not follow for internet applications. This shows that the design of the modelling environment should highly prioritize ease of use. King & Jun also found that studies that used TAM often did not require large samples sizes to get significant results.

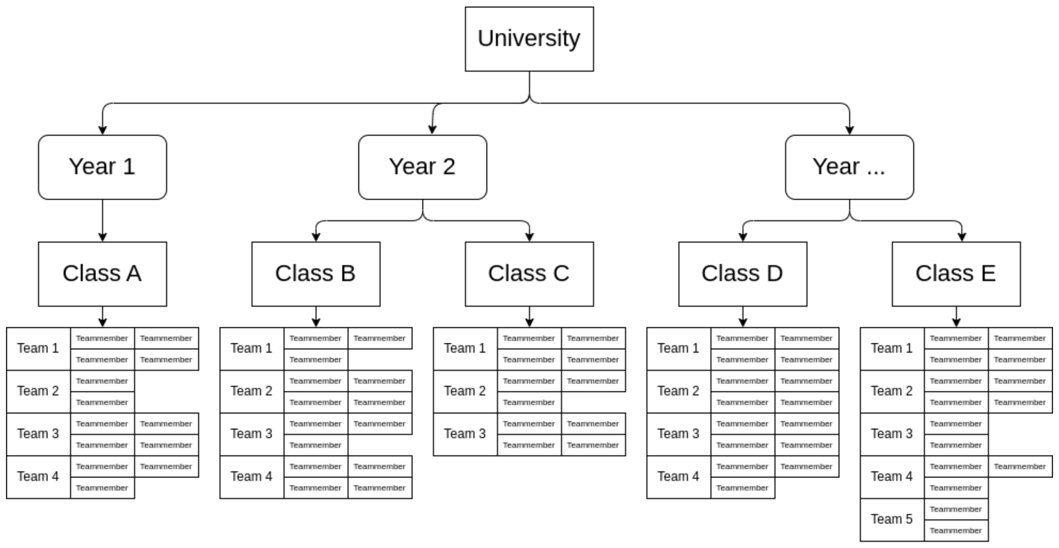
3 Architectural design

3.1 Overall design

This section will first introduce the intended users of the software and a short description of the developers working on the ngUML project. Then the concept of a component is introduced. Components are the way users will be able to share their work with others. Components will be discussed in detail, followed by an analysis of the desired server structure. The last part of the design section will focus on how real time collaboration should be implemented conceptually. For each design decision the problem it will solve is introduced, followed by a number of options that could solve it, and concludes with an analysis on which option is the best and should be implemented.

3.1.1 Users

The users of the ngUML software will primarily be university students who will use the software during a course. This means that most users will only have a few months to learn how to use the tools and to work with it to design a system. This also applies to the developers of the ngUML projects. Most of them will work on the software for the duration of a single semester or during their thesis. This implies that any functionality such as multi user collaboration and components should be very easy to understand. The developers do not have much time to learn to use intricate design patterns. As they will also need to work on new functionality, any protocols for concurrent collaboration should also be very easy to extend to new functionality. The same goes for the component system, that should also take into account that any future functionality should ideally be able to use the component system as well.



- **University**: The top user entity, the university sets up and owns the servers the software runs on. Inter-university collaboration would require shared servers. The university also manages classes.

- **Class**: The entity is owned by a teacher/lecturer/teaching assistant. Most collaboration will be between teams within a class. The class has it's own component library.

- **Teams**: A group of students tasked with modelling a certain part of a business. Teams may need to share common components, such as the representation of an employee. Either among their own diagrams or with other teams. Teams will also need to work together on the same diagrams, ideally by using a system which allows real time user collaboration.
- **Team member**: An individual within the team.

3.1.2 Component concept

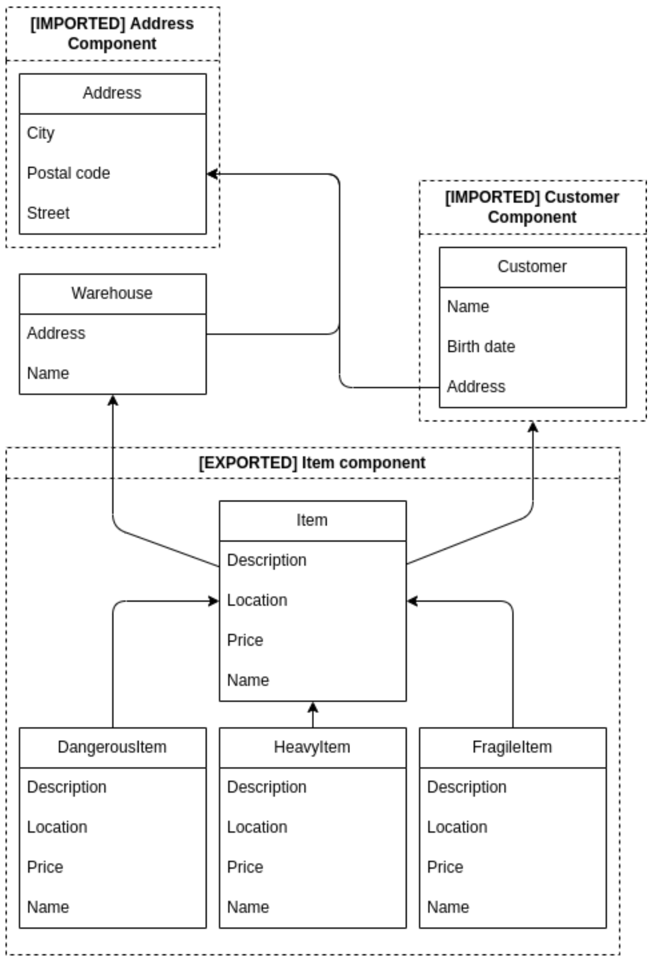
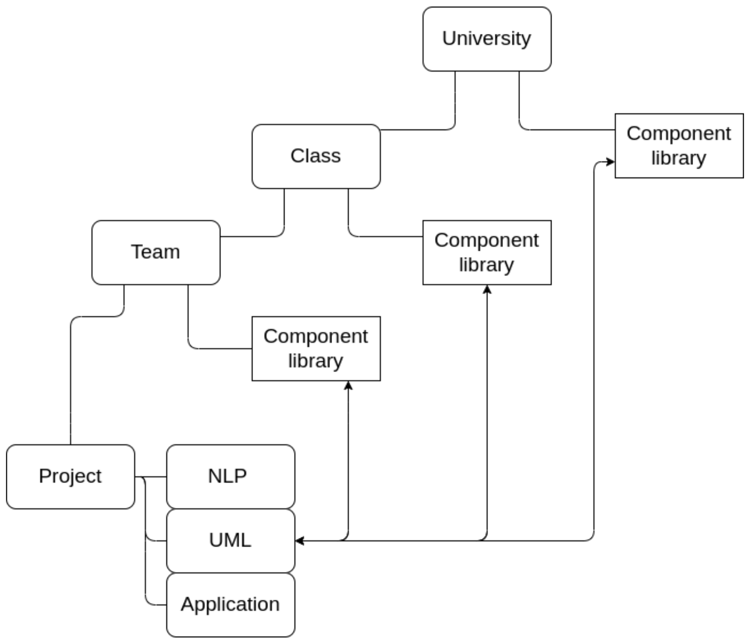


Figure 2: This diagram shows how a user might import two components from other teams and introduce his own. Schematic based on figure 5 from the paper [RGSC21]

In the figure you can view an example of a simple UML diagram, which shows how an order, a customer and a product relate to each other. These concepts are supported by some other entities such as delivery company and a line item. The boxes with a dotted line represent components. They are not part of the UML diagram and are added purely to visualize what parts of the diagram might consitute a component. For example this team is working on how to structure orders and products, and created an order entity and a product entity. They needed to ship the order to a customer, and another team already designed an entity for customer. They chose to not reinvent the wheel and imported the customer component from that team. Another team already needed an entity that represents a location, so they also imported the address entity. When another team might need to use product entities as well, the team selected the entity product and closely related entities such as dangerous product and restricted product. They then exported and shared it as a component with the other team. Now the team saved themselves time and their entities also align with what other teams within the class are using. No synchronization is needed.

A component is not restricted in size or complexity, it is simply a subset of an UML diagram. The user is free to choose which elements are incorporated into that component. A good component should be represents a part of a logical design, and therefore shares a lot of similarity to a class or function in programming languages. The design principles for well designed, easily reusable components are similar. It should be a natural compartmentalization of the UML diagram, and only represent a single thing. In the example above, if you did not import the user class, you would still need the address entity, to link the delivery company. If the creator of the user and address component only created a component in which both entities are present, you would have to edit the component and delete the user entity. In the case of the product component, the dangerous product and restricted product are so closely linked to the product entity, it would probably in most use cases not make sense to split those three entities into three different components.

3.1.3 Component library



- **Component**: A component is a (sub-)set of UML elements. There is no restriction in size, and it may consist of a subset or all the UML elements in the UML diagram of a design. However, a well designed component should be flexibly coupled to other elements and should be relatively self containing. Generally, it should only represent a single element. These guide lines enhance the reusability and are inherent to the compartmentalization of a logic system.
- **Component library**: A user can decide to create a component, and save it to a library. The user can also import components from any of which it is a member, to any diagrams.

The arrows show how components can be uploaded to libraries or imported from libraries. Component libraries do not have a fixed list of members, membership is decided by the creator of the library. He or she manually invites members or removes them. In the diagram an example structure is visible, where the university creates a general component library for all students of any course to export components to. The class instructor might create a library and add all class members to share components between teams. Teams will probably want to create their own library to share components between diagrams. The student can then import components into a UML diagram from any library he or she is a member of.

- **Project**: A project can consist of one or multiple UML diagram designs.

This structure is not only applicable to universities. It simply shows how the concept of component libraries can be used to share components with different groups of users. A company could create a single component library for the whole company, or create a component library for each department or for each project. It may also of course do both. This makes the design flexible, as it can be applied to any structure of users.

3.2 Component types

The component type dictates how you can import and export components, and how changes are communicated. It is important that it is intuitive to use and does not require too much manual input. Preferably the usability of components should not be impacted, as that will diminish the value of the importing and exporting system. It should empower the user to take full advantage of other people's components.

3.2.1 Options

- **Value components** The components are saved as an independent set of parts of the UML diagram. They can only be modified by manually updating the component itself. Any updates must be manually pulled in by the people using that model in their design.

- **Pointer components** The component refers to a zone or other delimitation in the design of the creator of the components. Any updates in the design within the zone are automatically updated to the component in the library, and any users of that component automatically receive that update too.

- **Hybrid components** The component must be manually updated, but any users of the update automatically receive the update. The advantage of this approach is that there is no need to implement anything like a zone in the design of the creator. It also avoids any confusion about what should be in the zone and what not. The link to the original component is broken as soon as the user of the component edits or deletes a part of the original component. The user may only move parts of the component around and link the component to it's own UML elements. This prevents edge cases.

3.2.2 Decision for UML Components

There are three requirements to consider when picking which type of component to implement for the UML phase. The first requirement is that it must be intuitive to use. A user should not have to invest much time to learn how to use components, as they are likely to only have a limited amount of time to work on the project. You can make component work flows intuitive by closely following existing work flows that are known to the average student. The second requirement is the usability of the component. This means that the component type must not impede the users work flow, and preferably enable the user to work faster. The third consideration is the development time required to implement a minimally viable component type. Due to the scope of a bachelor thesis, the implementation should not be too complex to implement and test within a reasonable amount of time.

The value component scores high on the first and third requirement. It is easy to develop and it is very intuitive for the user. It closely follows the copy paste work flow. You select the parts of the UML diagram that you would like to share, and it puts a copy into the library. To import it, you select the component in the library, and paste it somewhere in your own diagram. However, the usability is mediocre, as you need to reimport the component for each update from the creator of the component that you would like to use.

The pointer component scores high on usability, but is not very intuitive to use, and requires much more development time. You need to setup a system that shows the user what is currently considered to be part of the component, and then also enable the user to edit the component. That last step is not very complicated on the creator's side, but on the user's side it opens pandora's box of edge cases. You would need to develop intricate rules to decide how to handle any new, changed or deleted parts of the component after the user already linked and moved bits around of the original version of the component. So even though the usability of a pointer component is very high, as it requires no effort on the user side to handle updates, it also requires too much development time to implement.

The hybrid component scores high on intuitiveness, high on usability and requires less time to implement than pointer components. There are simple rules to avoid the edge cases on the user's side, and updating the component is very intuitive on the creator side. It is a good middle ground for all three requirements between the value and component type. Therefore the UML components will be implemented as a hybrid component.

3.2.3 Decision for text components

To also facilitate collaboration in the NLP phase, it is necessary to enable the sharing of text between projects and teams. This can be quite simple, by using the value component model. This component has the advantage that it is quite simple to implement, and it is very clear and intuitive for the user. Simply select the text that needs to be part of the component and create the component. Importing the component would be similar to pasting text from the clipboard in a program such as Microsoft Word. A pointer component would be quite complicated to implement, as that would require intricate rules when the end user starts editing the component. An hybrid component is easier to implement on the creator side, but the problem on the user side remains.

Considerng the aforementioned three requirements, the intuitiveness of a value component for text is high, the usability is also quite high, and it requires the least development time. In comparison to the UML components, the usability of value components for text is higher. Therefore it is an economical choice considering development time to aim for value components for text. In this thesis the text components will however not be implemented, due to scope restrictions. It was discussed here to show how other parts of the ngUML project could reuse this system of components in future functionality.

3.3 Handling external links

Another separate issue is the handling of external links from the subset of the UML diagram that will form a component to the other elements in the diagram. The employee class might have an address class connected to it that contains the city, street and postal code. The user would like to upload the employee as a component, but users will provide their own address class. This is an external link to a class that is not part of the component. Or, another use case might be that the user can choose whether to implement the address class himself or use the address class in the creator's UML diagram. There are multiple ways to handle this. The solution must indicate to the user that there is an external component that it needs to be linked to, it must be intuitive to use and it must not be too complex to implement.

- **Separate components**: The creator simply creates two individual components that are not linked together. Communication between the creator and the user is required to indicate that the two components are related. The users can then link the two components in their own diagram to link the two individual components, as external links are not broken when a component is updated.
- **Separate, linked components**: The creator either manually indicates that two components are related, or it is automatically evaluated whether the external link from a new component is to a component that is already in the user's component library. The user can then choose when importing the component to also import the related components.
- **Multi-layered components**: The creator creates a single component that encompasses both components, but makes subsets of UML elements within the component. The user of the component can then choose which subsets to import from the component.

The first option is the simplest, and very intuitive to use, as it is very clear what happens when you import or export a component. The user does not need guidance to learn a new system of selecting components. The second option requires some functionality to link components, or a simple algorithm to evaluate whether a component is already in the component library. This makes it slightly more complex. The multi-layered approach is less intuitive and much more complex. The user needs to understand what layers within a component are, a separate UI element must be developed to show the layers of the component and documenting or updating the component requires more complex. It also opens up more edge cases to handle in the user's diagram when the component changes.

The minimally viable product only needs to implement the first option, as the creator can also indicate in the description of the component that the user might also be interested in another component. The creator will always have to supply some short documentation on external links and the requirements of those external links. If the components are coupled so closely together that the user must use the other component too, it should probably be constructed as a single component, and not two separate components. If needed, a UI element may indeed be created to allow the creator to list related components and another UI element to select which components the user would like to import. This requires more complex for something that might not even be very intuitive to use and support a real use case. Because another solution to this problem is to not only upload the two components separately, but also upload another component that encompasses both. This is even more advantageous as the user does not have to manage any links between the two components, and the links can easily be updated by the creator. Therefore this project will implement the first option.

3.4 Composing a component

This design choice focuses on how a user can compose a component when working with the UML diagram. The selection of a component should be intuitive, and any changes to the component should be easy to communicate to the users of the component. Two options will be considered, a simple solution that closely follows existing design workflows, and a more innovative approach.

3.4.1 Click and select solution

The user can select items via a selection box. The user can then select and deselect additional items via individual CTRL + mouse clicks, or by expanding the selection box. This selection of elements then forms the new component. The only elements that are clickable are classes. Any links between selected classes will be brought into the component, but any external links are left out.

This approach is quite simple and intuitive for the user. Users are familiar with this as it is a very common pattern in many applications, such as file explorers and for example Microsoft's Excel. It is also easy to implement and works with the existing codebase in the frontend. It is also very clear what is and is not part of the component, while it is also very easy to change that selection.

3.4.2 Box implementation

In this approach, the user creates a box in a UML diagram. The box itself is the component space, everything within the box is part of the component. To remove an element from the component, simply drag it outside of the box. This approach could be used to enable automatic updating of the component, as it could update itself whenever the contents of the box change.

So the advantage of this approach is that allows easy modification of components. This could be useful at times when a component is still subject to many modifications, but is also used by other teams. The box approach is mostly relevant for pointer components, as value components do not benefit from the automatic updating. The disadvantage of this approach is that it is more complex to implement, and it inherently changes what a UML diagram consists of, as it adds another element. The user is also not used to this box approach, and that makes it less intuitive. It is also less flexible when moving elements around in the UML diagram, as all elements need to be within the box. It also requires all the elements to be positioned in a square box.

3.4.3 Decision

The first solution is significantly less complex to implement, more intuitive for the user and does not dictate the design of the diagram. The box approach is therefore not a viable approach.

3.5 Component library UI

The component library is kept quite simple, and will closely follow the design of a regular file system UI, as the components are just files within the component library. Each component will have some properties, such as owner, creation date, last modification date, title and description. Because of the relatively short time during only part of the year that a class uses the software, it is unlikely that a team will create a large number of components that need another system or UI. The implementation of folders does not belong to the minimally viable product, but is high on the priority list and the component class will be designed to be extensible to support folders.

3.6 Using an external hybrid or pointer component

Imported elements are updated automatically, and this has a few implications. You cannot freely modify the elements within the diagram. Allowing this would require an intricate system to transform modifications in the original import to something sensible in the newly updated component. External links by the exporter are ignored, and the importer may create external links. These external links are mapped onto the updated elements if the names match. If an element is deleted that had an external link, the external link is not imported. Elements can be freely repositioned by the importer. When an element of the component is deleted by the importer, it will be imported again on new updates. This is done to enable adding new elements to a component, without needing to compare what changed from previous component versions. We can simply do a comparison of what already exists in the diagram and what does not. It is possible that we will choose to not override a modification, but simply delete the link to the component. The element will no longer be updated when the component is updated.

3.7 Server structure

This section will review three options for the server structure that will run the website. Using multiple servers will allow the project to spread the workload on multiple servers, increasing the maximum load. It can also make the management of the servers easier. There are roughly three design directions that are feasible, that are listed below, coupled with the advantages and disadvantages of each approach.

3.7.1 Single server system

There is a single server that supports both the diagrams and the component libraries. The server is setup by the university, or by a hosting party if inter-university collaboration is required. This avoids a layer of complexity by using a monolith approach. This is especially useful as most people working on the project need to quickly be able to work on the project. It is more easy to maintain, but it scaling it might prove difficult, as dividing the workload on the server is harder. This is however negligible when you consider that it is unlikely for more than two classes per university to be using the system at the same time. There are most likely also much better options within Django for dividing the workload to multiple servers.

3.7.2 Multi server system

We leverage the concept of Django applications to spread the workload. This closely resembles a microservice architecture, where each application handles a specific part of the system. In this case one server could be hosting the components libraries, where only very basic CRUD operations are required. This server can be used over multiple years. Then you also have a server that hosts the projects and diagrams (the already existing backend). You could even spread the workload by using one server per class, while still using a central component server for all users. This server would only need to be used for the duration of the class. The component server can be quite simple, and reused over the years. Individual teams do not need to set anything up, and do not need to know anything about docker. You need two servers. You need a good server to support the workload of a class.

3.7.3 Decision

The two server structure seems to be the best option, as it is the more flexible approach. With the Django application system you can run multiple applications on a single server, but also spread them out on multiple servers. By designing the component library as a separate application we do not add any extra complexity, but even make the code better, as it is a clear separation of tasks. And as mentioned previously, this approach also allows for flexibility in the number of servers. Therefore the separate application, or service, approach is the one that will be implemented.

3.8 Real-time collaboration - Server

To allow users to see the actions of other users while collaborating on the same diagram, it is necessary for the server to adopt an approach on how to communicate these actions to the server, and back to the other users. There are three general approaches that we can take to communicate these changes.

- **Polling**: This is the most simple approach. Each client simply polls the server periodically for any changes to the shared model. It does not require any external libraries, and the pattern is very simple and intuitive. A form of long polling can be simulated on the server side.
- **Long-polling**: It performs nearly as well as websockets ([PN12]), and it is a very intuitive pattern. It requires the Django Channels library.
- **Websockets**: It performs the best of all three options, and can handle large volumes of requests per client. It requires the Django Channels library. It does use a different protocol than HTTP, which will require more knowledge on the programmer's end.

The websockets option would be the best approach in terms of latency, but requires another external library and this optimal latency might not be necessary. As long as the amount of requests and required latency is relatively low, it is not necessary to implement websockets. The long polling approach is better than polling in terms of latency, but also requires programmers to learn another external library. The best option is polling, as it suits the project does not require very low latency, is easy to implement and it is also easy to work with. This is optimal for the ngUML project, as quick onboarding of potentially inexperienced developers is required. Another significant advantage of polling is that the stress on a server could very easily be lowered by setting the polling interval higher.

3.9 Real-time collaboration - UI

To enable the communication of user actions to other collaborators, there are a plethora of approaches. Three approaches will be considered, that differ in usability, difficulty of implementation and intuitiveness.

- **Button**: A button will be added to the UI that will enable edit mode for the user. Other users will be locked out of editing the diagram, and they will see a grayed out lock button. After the first user is done editing, he will click on the button again to give up control of the edit mode. Other users can now take over the edit mode by clicking on their button. During read-only mode, users can still compose components, and import those to another diagram to work on them, and then reimport it into the main diagram when they have edit control. Updates by the editor may be shown to other users in real time or be updated after the user leaves edit mode.
- **Open-mode**: The diagram page will be opened in read-only mode or in edit mode. The edit modus works similar as in the button design, but while the user is in edit mode, other users can only open the diagram in read only mode, and need to reopen the page in edit mode to gain edit control.
- **Real-time feedback**: All users can edit the diagram at the same time, and can see the changes of their team members as they are being made.

The third option will require more requests and thus a better server. It might even not be feasible to run for multiple teams at once on a single server, depending on how often the changes are communicated to collaborators. The first and second option are however more flexible in that approach, as they can delay any updates until the editor is done. The first option is more user-friendly, as it allows quicker switching between edit mode for users, because they do not have to reopen the page. The program can then just use polling to both get the latest changes by the current editor, and also detect when the diagram is unlocked, and available for someone to edit. It is also far easier to both implement and expand on, which is why that is the most suitable approach for the ngUML project.

4 Implementation

In this section the actual implementation of the design will be discussed. The implementation needs to integrate with a large existing codebase, so the code should be easy to integrate. As previously mentioned, most future developers who will work with the code do not have much experience. They also do not have much time to familiarize themselves with the codebase, as they need to deliver their work within the span of a few months. So besides the requirement of needing little integration work, it should also be easy to understand and easy to extend, to help future developers.

The implementation of the design can be split into two sections. The first section will discuss how multi user collaboration on a single diagram was implemented. This will go into detail on how the server guarantees that only a single user at a time can edit the diagram via a locking system. The second section will discuss the implementation of components. It starts off with the implementation of components in the database, then it will discuss how diagram entities are linked to components, and after that it will discuss how components are synchronized to recent versions. This part is the part that implements the concept of hybrid components as designed in section 3.2. The last part of the implementation section will discuss the library UI view, which allows users to add and remove users to the library, manage components and edit the properties of the library.

4.1 Lock implementation

As discussed in the design section, to enable multiple users to work on the same diagram a read only and edit mode was implemented. As seen in figure 4.1, this is communicated to the user via a button that has three states. The three states refer to the ownership of the lock. In the first and default state, the lock is in use by no one. Any user may click on the button displaying the text 'lock' to acquire the lock. In the second state, the user has acquired the lock and the lock button displays the text 'unlock'. When the user now clicks on the button, it releases the lock so someone else can acquire the lock. In the third state, the user can not edit the diagram as another user has acquired the lock. The lock button now displays 'locked', and clicking it will refresh the lock state. This can be used by the user to check whether the lock was freed by another user.

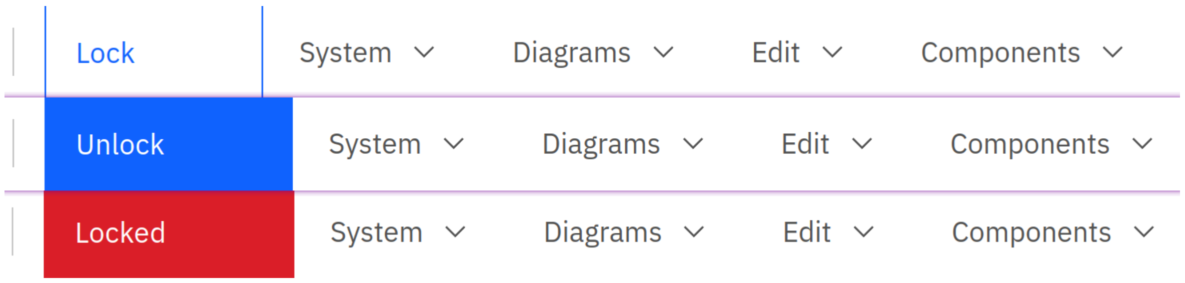


Figure 3: The three states of the lock button in the top bar when viewing a diagram. Lock states in order from top to bottom: free, in use by current user, in use by another user.

The lock state does not only change when the user clicks the lock button. When the lock is free and the user starts editing the diagram, the lock is automatically acquired. Every time the diagram is refreshed, the lock state is refreshed as well and checks whether the lock expired. The lock ownership currently automatically expires 1 minute after the last edit was made. So each time a user who owns the lock edits the diagram, the lock expiration is reset. This value is configurable, so it could for example also expire after 60 minutes. This value influences how the users can interact with a diagram. A short lock expiration will allow users to switch control often, simulating more of a fine grained user collaboration experience. A longer expiration will simulate the workflow of for example older version control software much more. Where users check files out and only check them back back in after they are completely done with their edits. This could be simulated by setting the lock expiration to 60 min, a user can take his time to make all his edits, and when he is done he can either let the timer expire or release the lock manually.

To implement this lock, two fields were added to the Model object in the database. This Model object constitutes the data related to the diagram in the frontend. It inherits from `models.Model`. The class `models.Model` is a class from Django which informs Django that the class should be represented in the database. The first field that was added is the `last_lock_interaction_time`, this is a timestamp indicating the last time that the lock ownership changed. The second field that was added is the `lock_owner` field. This field indicates which user currently owns the lock. If it is blank, it is not owned by anybody.

```python
class Model(models.Model):
  ...
  last_lock_interaction_time = DateTimeField(auto_now=True)
  lock_owner = ForeignKey(User, on_delete=models.SET_NULL, null=True)
  ...
```

Not only should the locking system be simple to use for the users, it should also be easy to integrate into the existing system. Ideally it should also be flexible for any new functionality. That is why the lock validation is implemented as a python decorator function. In short, a python decorator is used to execute code before or after executing another function. To do this you first write a decorator function and then decorate the function you would like to modify with it. In the function below a decorator is implemented that can be used to decorate any of the diagram API functions. It checks whether the lock is in use, and if it is whether it either expired or is owned by the user making the request. If not, it returns an exception. If the lock is free, it acquires it for the user, if the lock was already owned, it updates the time stamp of the lock.

```python
def concurrency_lock_protected(func):
  """Checks if the acquirer can access the lock before the function is executed"""
  @functools.wraps(func)
  def wrapper(*args, **kwargs):
      model = args[0].get_object()
      acquirer = args[1].user
      now = datetime.now(timezone.utc)

      is_owned_by_other = (model.lock_owner is not None
                           and model.lock_owner != acquirer)
      lock_expired = now - model.last_lock_interaction_time > LOCK_DURATION
```

```python

        if is_owned_by_other and not lock_expired:
            raise exceptions.APIException(f'Diagram is locked.')

        model.lock_owner = acquirer
        model.last_lock_interaction_time = now
        model.save()

        return func(*args, **kwargs)

    return wrapper
```

Below you can see the signature of one of the many diagram API functions. This one in particular is used to reposition nodes along the x and y-axis in the diagram.

```python
@decorators.action(detail=True, methods=['PUT'])
def reposition(self, request, *_, **__):
  ...
```

All we have to do to adapt this function to the new locking system is decorate it with the decorator function above.

```python
@concurrency_lock_protected
@decorators.action(detail=True, methods=['PUT'])
def reposition(self, request, *_, **__):
  ...
```

Now the function can only be called when the lock of the underlying diagram is free or in use by the user making the request. The decorator can and should be applied to any function that is modifying the diagram. It does not require any changes to the body of the function or knowledge from the programmer about how to call the function. The decorator function only requires that it is used on a diagram API request, so it can read the diagram ID and user ID from the `request` parameter.

The locking system also implements another decorator. This decorator evaluates whether the lock expired, and if so it sets the lock ownership to `None`. This decorator does not check whether the user making the request owns the lock. This decorator can be applied to any API functions that do not modify the diagram, and so do not require lock ownership, but only read the diagram. Currently this decorator is applied to the API function which loads and refreshes the diagram in the frontend. This allows users to see when the lock is available to acquire each time the diagram is refreshed in their UI.

```python
def check_lock_expiration(func):
  """ Sets lock state to None if it expired """
  @functools.wraps(func)
  def wrapper(*args, **kwargs):
      model = args[0].get_object()
      now = datetime.now(timezone.utc)

      lock_expired = now - model.last_lock_interaction_time > LOCK_DURATION

      if (lock_expired):
          model.lock_owner = None
          model.last_lock_interaction_time = now
          model.save()

      return func(*args, **kwargs)

  return wrapper
```

4.2 Component model

The components and component libraries were also implemented as Django database objects. The class below is used to represent the concept of component libraries. Each library has a name, a description and a creation date. It is also is connected to a collection of users. These users are in the list of members of the object. One of the users is also the owner of the library. At first this is the creator of the library. If the creator leaves, one of the members is promoted to owner.

```python
class ComponentLibrary(models.Model):
  name = CharField(blank=False, max_length=200)
  description = TextField(blank=True)
  owner = ForeignKey(User, on_delete=models.PROTECT)
  date_created = DateTimeField(auto_now_add=True)
  members = ManyToManyField(User, related_name="membership",  blank=True)
```

The componentblob is slightly more complex. This object is used to represent the concept of a component. It was important that the component system and libraries could also be reused for any future functionality, that is why the componentblob object is completely agnostic to it's content. The only requirement is that it the contents of the component comprises of valid JSON. This way it can later be reused to store for example natural text that was used to describe systems to the UML generating system. The other interesting fields of the component is the version number field. As the component synchronization code relies on valid and consistent version numbers, the version number of a component is automatically increased when it is changed. This prevents bugs relating to version number not being updated by other parts of the codebase even though the componentblob in fact did change. This auto-incrementation is done by over riding the `save` function of the parent `models.Model` object. The last modified timestamp acts in a similar way, updating itself each time the componentblob is updated.

```python
class ComponentBlob(models.Model):
  name = CharField(blank=False, max_length=200)
  description = TextField(blank=True)
  content = JSONField()
  version_number = IntegerField(default=0)
  date_created = DateTimeField(auto_now_add=True)
  date_modified = DateTimeField(auto_now=True)
  owner_library = ForeignKey(ComponentLibrary,
    related_name="owner_library", on_delete=models.CASCADE)

  def save(self, *args, **kwargs):
      self.versionNumber += 1
      super().save(*args, **kwargs)
```

4.3 Component metadata

To be able to update components within a diagram, information needs to be stored about the origin of each entity within the diagram. So if it was created by importing a component, it should also keep some data about which component and which version it was imported from. The diagram contains several different entities, and each entity should hold the same information regarding it's origin. To also implement a generic and flexible solution for this problem, three generic API methods and a mixin was implemented. The goal is similar to the lock implementation, where it should be easy to integrate with the existing codebase and also be very easy to use with future functionality.

A mixin in Django is a class which inherits from `models.Model`, and adds a few data fields. Any diagram entities then instead of inheriting themselves from `models.Model`, inherit from the mixin. The mixin is also defined to be abstract. This causes Django to add the data fields to the child that inherits from the mixin, while the mixin itself is not represented in the database. In short, it allows programmers to define the same properties on several database tables, which in this case are the diagram entities.

The mixin is called `ComponentMetaData` and consists of three fields. The first field refers to the component the entity was imported from, the second field indicates the version of the component and the third field indicates whether the connection to the component is still active. If this field is inactive, it indicates that it should not be updated when the user synchronizes the component entities in the diagram. The first benefit of setting inactive to true instead of setting the metadata to zero, is that it is still possible to view the origin of diagram entities. The second benefit is that when a component is synchronized, it can match the inactive element with an element in the component, and decide to not add the new version of the entity, while also not deleting the old entity. It can remain out of scope of the synchronization.

```python
class ComponentMetaData(models.Model):
  component = ForeignKey(ComponentBlob, default=None,
    null=True, on_delete=models.SET_NULL)
  component_version = IntegerField(null=True, default=None)
  component_active = BooleanField(default=True)

  class Meta:
      abstract = True
```

To showcase how easy it is to integrate with the existing database entities which make up the entities in the diagram, below is the signature of the old `Relationship` class. This class is the entity which represents connections between classes. It currently inherits from `models.Model`.

```python
class Relationship(models.Model):
  ...
```

By changing the parent to the mixin, it adds all the fields from `ComponentMetaData` and now indirectly inherits from `models.Model`. This shows how non-intrusive it is to change existing diagram entities in the database to add componentmetadata fields.

```python
class Relationship(ComponentMetaData):
  ...
```

The mixin allowed the implementation of component metadata to be generic on the database side. The API however also needed to be adjusted to pass information to the frontend about the meta data. There are two options to do this. Either every existing API method is updated to also include component metadata in each GET, POST, PUT and PATCH request, or three separate API requests are constructed which allow modification of the component metadata for each diagram entity. The latter option is obviously preferable, as it is less intrusive for existing entities and protects any new programmers from forgetting to pass and allow modification of the component metadata.

So three API methods were introduced, the first method allows for the retrieval of the component meta data, the second allows modification of the component meta data. Logically the third method should be able to delete the component metadata. But deletion in this case is called reset, as all the data is just set to null to indicate that the entity no longer originates from a component. For inactivating a component, the second API method should be used. All three methods take at least two arguments, the first is a string indicating the type of the entity, and the second argument is the id of the instance of the entity. With these two arguments it is possible to write the three API methods that can be used for any of the diagram entities in a generic way. Simply pass the arguments to the helper function written below and it returns the correct row from the database. It does so by using a dictionary which maps the strings to the correct database objects. When a new diagram entity is implemented, the programmer only needs to make sure it inherits from the mixin and he or she should add a new key to the mapping.

```python
def get_object_for_id_and_type(self, object_id, object_type):
    from model.models.classes import Class, Enumerator,
      Property, Operation, Association, Generalization, Composition

    type_db_class_mapping = {
        "class": Class,
        "enum": Enumerator,
        "property": Property,
        "method": Operation,
        "association": Association,
        "generalization": Generalization,
        "composition": Composition,
    }

    db_class = type_db_class_mapping[object_type.lower()]
    if db_class is None:
        raise exceptions.APIException(
          f'{object_type} is not a valid object_type.')

    return db_class.objects.get(id=object_id)

```

4.4 Component synchronization

Component synchronization allows user to update the components in their diagram to the latest version of the component published by the owner of the diagram. This synchronization uses the component meta data and the related API functions discussed in the previous section to synchronize the components. First the functionality of the synchronization is shown via five screenshots of imported components. After that the technical implementation is discussed on how this synchronization was achieved and how it was made extensible for any future functionality.

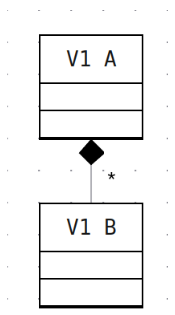


Figure 4: In this basic scenario, the user imported this simple component from a component library.

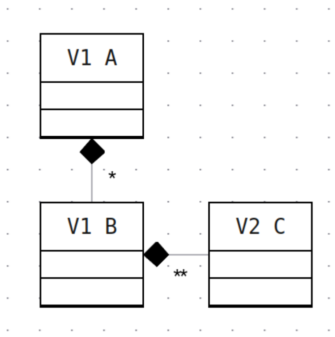


Figure 5: The owner of the component updated the component with an extra node. Synchronizing the component will automatically add this new node to diagram of the user of the component.

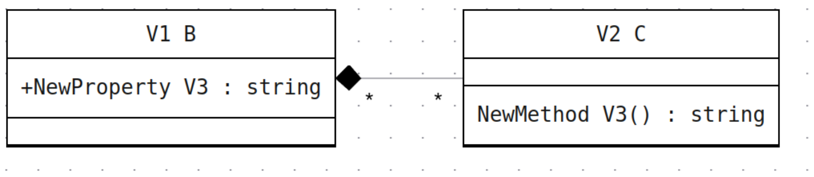


Figure 6: Not only newly added nodes are synchronized, but in this scenario the owner decided that the 'V1 A' node was no longer necessary, and thus removed it from the component. He also added a new property to the 'V1 B' node, and a new method to the node added in version 2, the 'V2 C' node. All these changes were also synchronized to the diagram of the user of the component

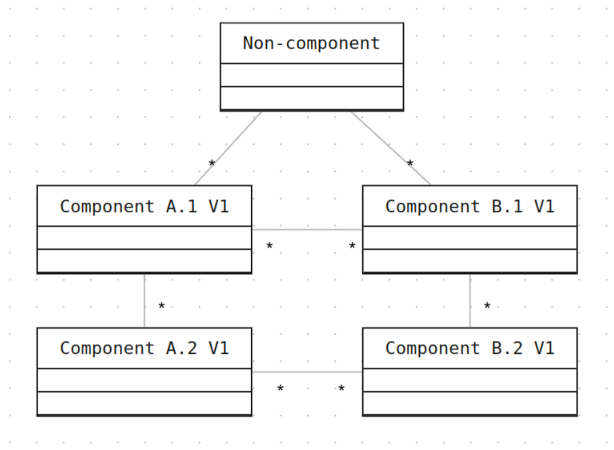


Figure 7: Component synchronization also works when the component is connected to other diagram nodes, or even components.

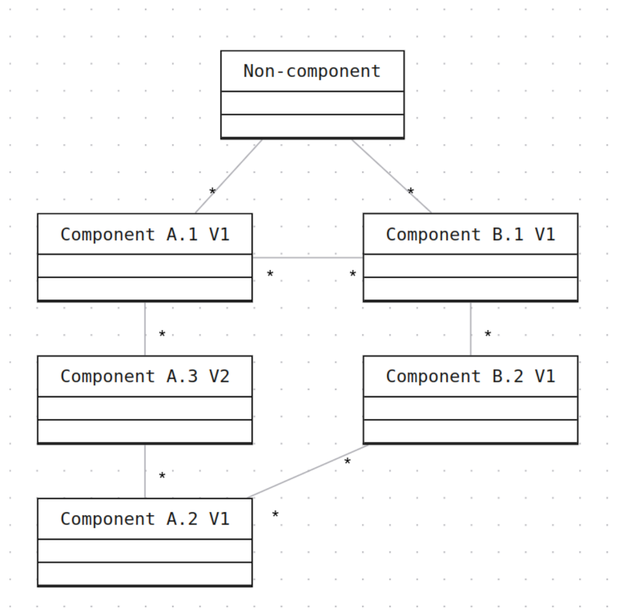


Figure 8: The owner of component A realised that actually a third node was needed, which acts as a layer between node 1 and 2 of component A. After the user synchronized the component, all connections to non-component nodes and component nodes remain unchanged, while the new node was seamlessly added to the diagram. When connected component nodes have changes in their properties or attributes, these get synchronized as well, without removing any connections to the node.

The synchronization process cannot be made completely generic, as it needs to connect to existing API functions which are tied to each type of diagram entity. Large parts of it are agnostic to what is actually in the diagram, so it is very easy to port the synchronization process to new types of diagrams in the future, or extend it to new diagram entities.

The first step of the synchronization process is to collect all nodes and edges in the diagram into a list. This list is then filtered to only contain nodes and edges which are part of a component (even though they may be inactive). This list is then split into a list of nodes and edges for each unique component. Then for each set of nodes and edges specific to a component, the latest version of the component is retrieved via the API. The set of nodes and edges related to the component, and the latest version of the component is then passed to a synchronization strategy function.

A synchronization strategy function is simply a function which will synchronize the diagram entities to the component according to a certain set of rules. This separation between the collection of nodes and edges related to the component and updating them is done in order for future extensibility. It is very easy to implement a new synchronization strategy in the future, simply implement the strategy and hook it up to the code that collects the nodes and edges per component. A new strategy might be implemented to give the users the choice in what way components will be updated. A more conservative synchronization might for example be implemented that conserves old nodes, or nodes that contain changes made by the user of the component. This might offer users more control in the future in the synchronization process.

The synchronization strategy that was implemented for this thesis is very easy to extend if new diagram entities are added in the future. It uses a generic function that maps diagram entities to their component counterparts. It divides all the entities that are passed to the function in three different lists. The first list are the diagram entities which could be matched with a component diagram. The list contains tuples of the matched diagram and component entities. The second list contains all component entities which could not be matched with a diagram entity and the third least does the same thing, but then for diagram entities.

Below you can view the signature of the function. It takes a list of component entities and a list of diagram entities. This function is currently used four times. It first maps all the diagram nodes to component nodes, it then is called again to map all the diagram edges to component edges. For matching nodes, the function is called once again to map all diagram properties of the node versus the properties of the node in the component. The same is done for the methods of matched nodes.

```
function mapEntities(componentEntities, diagramEntities,
                     identificationFunction, identificationArgStruct) {

  ...

  return {
      matches: matchedEntities,
      onlyInComponent: onlyInComponentEntities,
      onlyInDiagram: onlyInDiagramEntities,
  }
}
```

The third argument of the function is the `identificationFunction`, which is the key to allow this function to be generic. This function is called on each component entity and diagram entity. If the id it generates for a diagram entity and it's component counterpart is equal, the `mapEntities` function will consider the entities to match. If a diagram or component cannot be matched based on the id, the entity is added to the `onlyInComponent` or `onlyInDiagram` list. Some identification functions are quite simple, such as the function below which will match properties in the diagram to properties in the component based on their name.

```
function generateIdFromProperty(property, isPartOfComponent, argStruct) {
  return property.name
}
```

The identification function for edges is more complex. Edges do not contain a property which can easily map the edge to an edge in the component. To match edges in the diagram to edges in the component, a comparison based on the two nodes it connects is done. However, the edge does not store the name of the nodes it connects, only the id. As the id of diagram nodes and their component counterparts is different, their respective id's need to be matched to their names first. It then generates a string which contains the name of the source node and the name of the target node, which are identical for both diagram and component edges.

To facilitate this more complex identification, the `mapEntities` function passes the argument `identificationArgStruct` to the identification function. The boolean `isPartOfComponent` is also passed, which indicates whether the identification function is identifying a diagram entity or component entity. In the edge identification function this is used to decide which of the two mappings that were passed via the `identificationArgStruct` should be used.

```
function generateIdFromEdge(edge, isPartOfComponent, argStruct) {
  let nodeIdToNameMapping = undefined;
  if (isPartOfComponent) {
      nodeIdToNameMapping = argStruct.componentNodeIdToNameMapping;
  }
  else {
      nodeIdToNameMapping = argStruct.diagramNodeIdToNameMapping;
  }

  return `${nodeIdToNameMapping[edge.source]}:${nodeIdToNameMapping[edge.target]}`
}
```

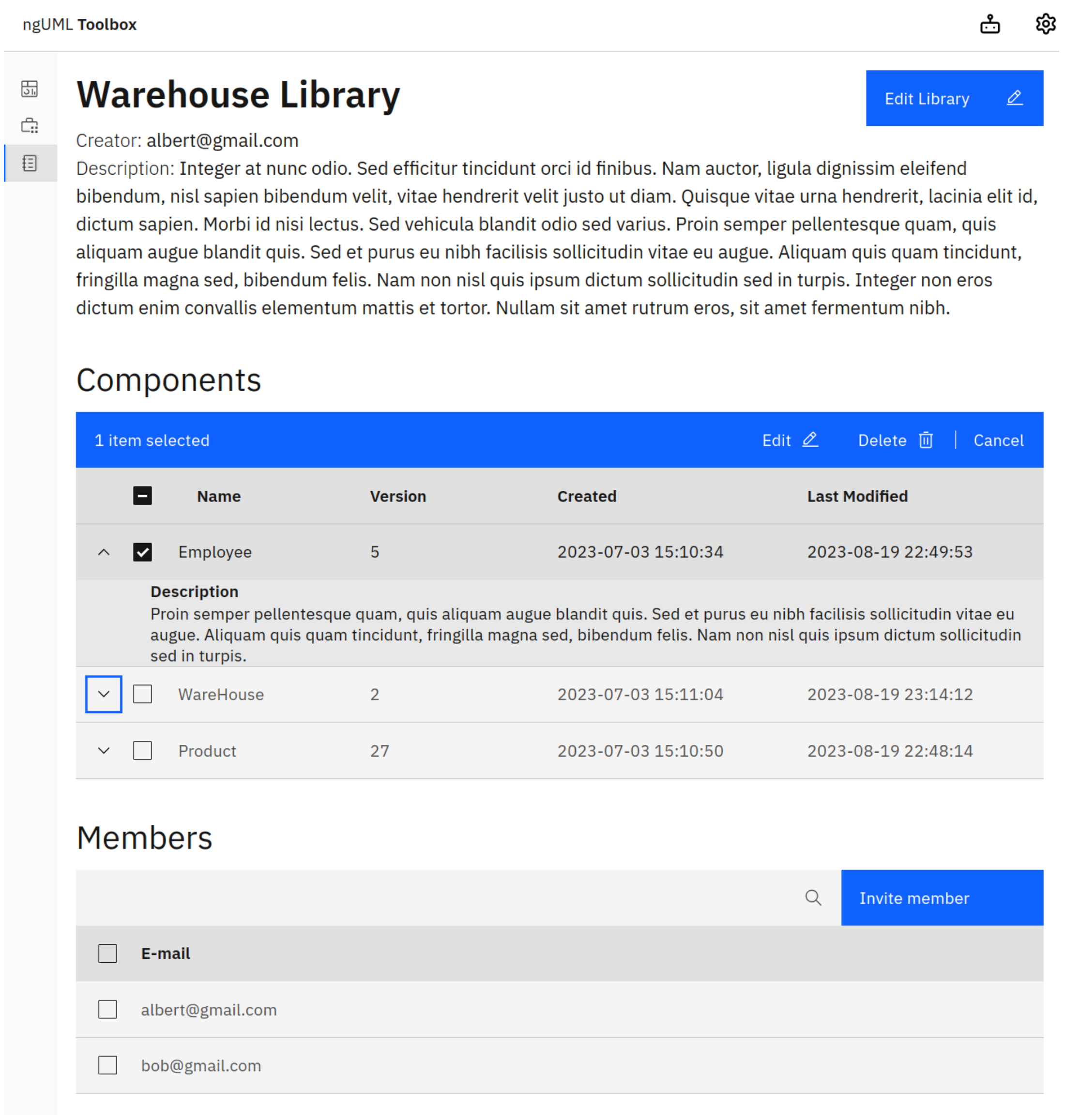
With this generic mapping method, it is as easy as generating the mapping and then resolving each item in each of the three lists. Below is a code block showing how each item is passed to a function that will handle the entity according to the rules of the synchronization strategy. For the strategy implemented in this thesis, the resolve functions mostly consist of a single API call to create, delete or update the diagram entity and update the component metadata. If a new type of entity is created in any future functionality, integrating it with the component synchronization is as simple as writing an identification function and writing three functions that resolve each item in the three lists that the mapping function generates. The synchronization strategy is not only generic enough to easily support new diagram entities, but it can also be ported to other diagram types in the ngUML project, such as use case diagrams and activity diagrams. It only needs to be connected to the API functions of the diagram type, and then an identification function must be written for each type of diagram entity. After that it is only a question of generating the mapping for each type of diagram entity and processing the result.

```
...

methodsMapping.matches.forEach(match =>
  resolveMethodMatch(match.component, match.diagram))
methodsMapping.onlyInComponent.forEach(method =>
  resolveMethodOnlyInComponent(method))
methodsMapping.onlyInDiagram.forEach(method =>
  resolveMethodOnlyInDiagram(method))

...
```

4.5 Component library overview

As shown in figure 4.5, the user can view the components in a library and the members of the library on the library overview page. The library overview pages also shows the description and creator of the library and allows the users to edit the library name and description. The rest of the overview consists of the component overview and the member overview. The component overview shows the name, the version number, the creation date and the last modified date of each component. By clicking on the component, a dropdown appears which shows the description of the component. By clicking the component, it is selected and then the name and description can be edited. By selecting additional components, multiple components at a time can be deleted. The overview also contains a search bar, to allow users to quickly search for the component they need in larger libraries.

The library overview also shows an overview of the members. This overview also contains a search bar and is the interface via which users can be added or removed from the library. Only members who are added to the library can view and add components to the library. By clicking on the 'invite member' button, new members can be added by entering their email address in the form that will show up.


ngUML **Toolbox**

Warehouse Library

Edit Library

Creator: albert@gmail.com

Description: Integer at nunc odio. Sed efficitur tincidunt orci id finibus. Nam auctor, ligula dignissim eleifend bibendum, nisl sapien bibendum velit, vitae hendrerit velit justo ut diam. Quisque vitae urna hendrerit, lacinia elit id, dictum sapien. Morbi id nisi lectus. Sed vehicula blandit odio sed varius. Proin semper pellentesque quam, quis aliquam augue blandit quis. Sed et purus eu nibh facilisis sollicitudin vitae eu augue. Aliquam quis quam tincidunt, fringilla magna sed, bibendum felis. Nam non nisl quis ipsum dictum sollicitudin sed in turpis. Integer non eros dictum enim convallis elementum mattis et tortor. Nullam sit amet rutrum eros, sit amet fermentum nibh.

Components

1 item selected — Edit — Delete — Cancel

	Name	Version	Created	Last Modified
✓	Employee	5	2023-07-03 15:10:34	2023-08-19 22:49:53
☐	WareHouse	2	2023-07-03 15:11:04	2023-08-19 23:14:12
☐	Product	27	2023-07-03 15:10:50	2023-08-19 22:48:14

Description (Employee)
Proin semper pellentesque quam, quis aliquam augue blandit quis. Sed et purus eu nibh facilisis sollicitudin vitae eu augue. Aliquam quis quam tincidunt, fringilla magna sed, bibendum felis. Nam non nisl quis ipsum dictum sollicitudin sed in turpis.

Members

Invite member

	E-mail
☐	albert@gmail.com
☐	bob@gmail.com

Figure 9: Screenshot of the library overview page.

4.6 Reflection

The implementation of all the features has slightly deviated from the design at certain points. It does however satisfy all the requirements that were posed in the design section. It supports multiple users working on the same diagram, and allows users to share their work to further enhance collaboration. Every part of the implementation was also highly influenced by the three key design aspects for this project: easy to integrate, easy to understand and easy to extend. This has certainly achieved by using common design patterns, leveraging Django functionality and the implementation of generic functions that can be reused.

5 Testing

5.1 Experiments

5.1.1 Quantitative Experiments

To test the impact of the locking mechanism on the server speed, a test will be done with a varying amount of users. The first test will be a collection of users each repositioning a random node in separate diagrams. In this test the locking mechanism will never need to block the repositioning, as no other users are working on the same diagram. The second test will be the same collection of users trying to reposition the same node in the same diagram. In the second test the users will continuously block other users, as each time they successfully reposition the node, the lock is acquired by them. Other users will then need to wait until the lock is released.

The users are simulated by a script written in Python. In this script, each user is assigned a separate process in which it continuously tries to reposition the node. If a user fails to reposition the node, the lock was not free. The user will then simply immmediately try to reposition the node again. To simulate users making a varying amount of requests before releasing the lock, a random element is introduced. Each time it successfully repositions the node (and thus acquired the lock), there is a 1 in 3 chance that the user decides it does not want to make any other changes and releases the lock. Otherwise the user will reposition the node again. If it decides it is not done yet, it will reposition the node again. After the first successful repositioning request, each subsequent repositioning request will always be successful until the lock is released, as the user acquired the lock in the first successful repositioning request.

The test was executed locally, so server latency is minimal. All users are simulated simultaneously on different processes, for a duration of 120 seconds. The tests were executed on a computer with 16 GiB of RAM memory and an Intel®Core™i7-7700K CPU running at 4.20 GHz. The processor has 4 cores, with two threads on each core for a total thread count of 8.

5.1.2 Qualitative Experiments

The user experience of using both the component system and the locking system was tested by conducting a survey. Due to scope limitations of the thesis, there was not enough time or resources to let the respondents work with the software for an extended period of time. The total number of people participating was nine. The ages of the users ranged from 20 to 55, and varying amounts of experience with UML diagram design. All users were comfortable with technology and have experience using software programs for either their studies or their job. Users were first taught how the diagram tool functioned and how they could map concepts and relationships onto a UML diagram. After that they were taught how to use the locking system and the component system. At the end, they were asked to fill in the questionnaire based on their experience with both systems.

The survey questions were based on the technology acceptance model questionnaire of Fred Davis, the creator of the technology acceptance model. Two modifications were made due to the background of the subjects and the nature of the ngUML project. As the ngUML software does not yet have an established user base, and the subjects of the questionnaire do not have experience with UML diagrams. Therefore the sections anticipated use and perceived characteristics were left out of the survey, as users could not possibly make an accurate judgement of that. The second modification was that some of the questions were rephrased to place less emphasis on using the software for their actual job, and instead use the more general word ’work’. This is done because most of the users did not use UML diagrams for their job.

All questions were answered by rating it on a scale from likely to unlikely, with 7 options: extremely unlikely, quite unlikely, slightly unlikely, neither, slightly likely, quite likely and extremely likely. Except for the question ’my usage of components in my work would be’, that one was answered on the same scale but with the words ’negative, positive’ on each end. A score of 1 denotes an answer of extremely unlikely/negative, and a score of 7 denotes extremely likely/positive. As the functionality that was implemented in this thesis could be divided into two distinctive parts (components and diagram locking), users filled in two separate questionnaires, where one focused on the usage of components, and the other on the usage of diagram locking.

5.2 Results

5.2.1 Quantitative Experiments

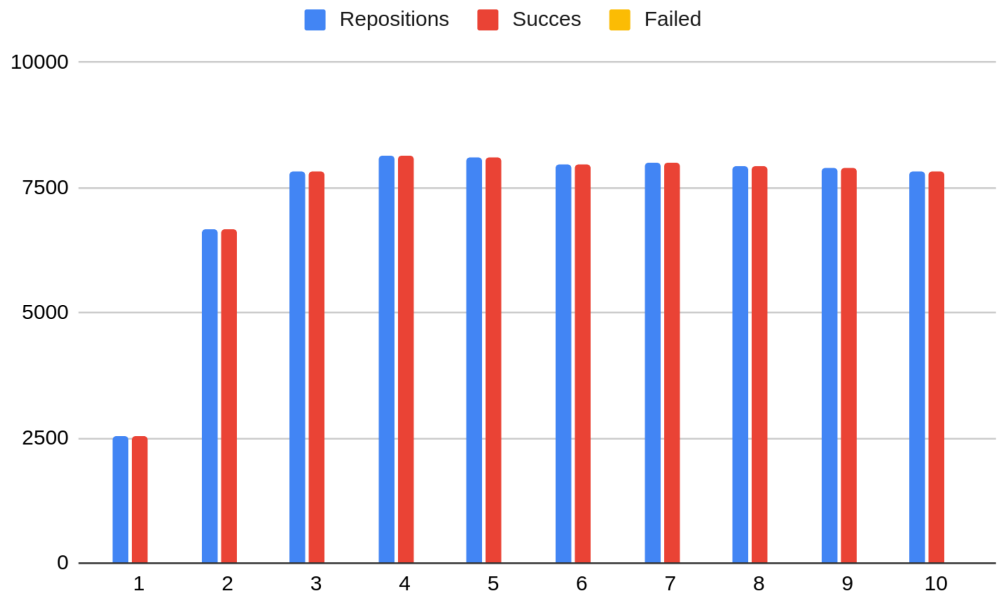


Figure 10: The total, successful and failed release requests done per run of 120 seconds with each time a varying number of users (as displayed on the x-axis) each repositioning a node in a separate diagram. The y-axis shows the number of requests.

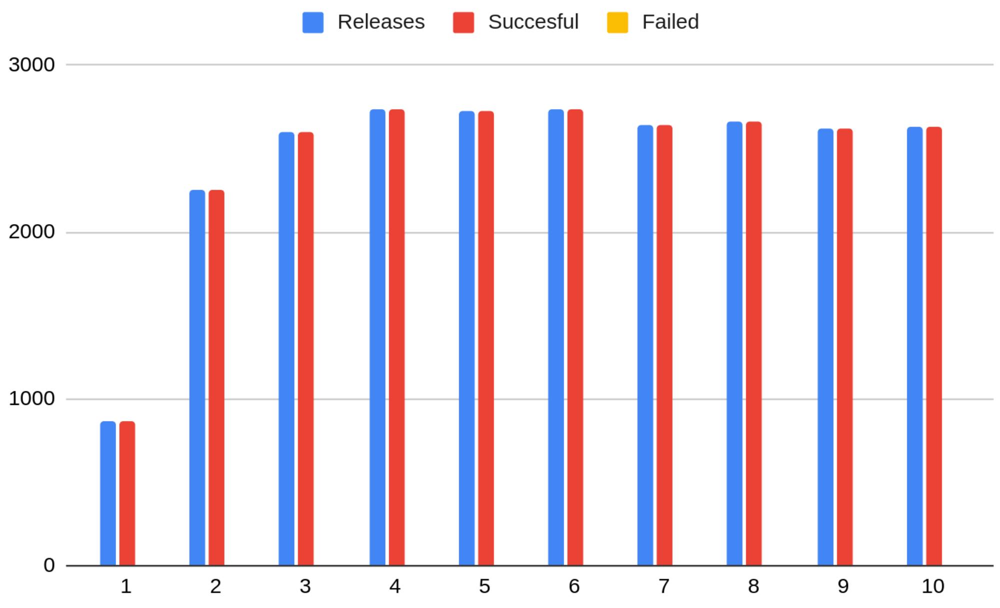


Figure 11: The total, successful and failed release requests done per run of 120 seconds. The number of release requests is proportional to the number of successful reposition requests. The y-axis shows the number of requests

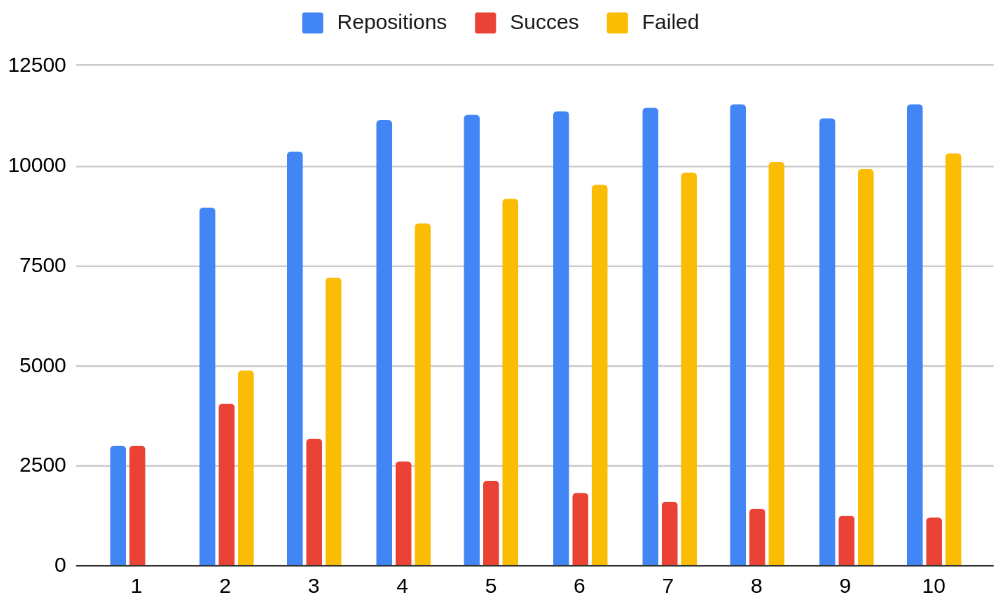


Figure 12: The total, successful and failed release requests done per run of 120 seconds with a varying number of users (as displayed on the x-axis) repositioning the same node in the same diagram. The y-axis shows the number of requests.

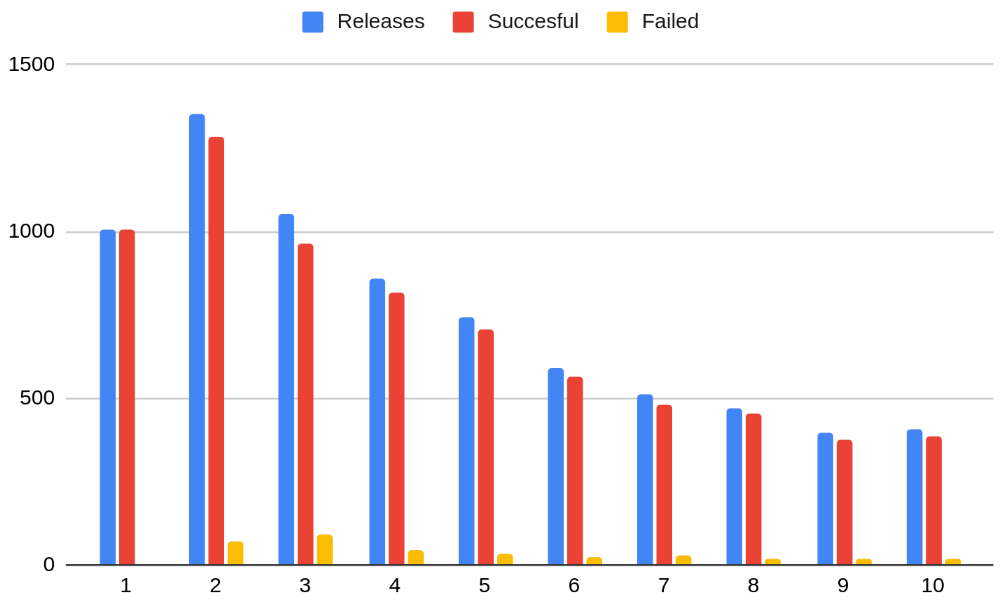


Figure 13: The total, successful and failed release requests done per run of 120 seconds. The number of release requests is proportional to the number of successful reposition requests. The y-axis shows the number of requests.

5.2.2 Qualitative Experiments

Components (n=9)	Average	Std dev.
General		
How confident are you in the ratings that you have made?	4.63	1.51
My usage of components in my work would be [negative, positive]	5.25	1.04
I predict that I will use components on a regular basis in the future	4.63	1.06
Perceived ease of use	3.90	1.59
Learning to use components would be easy for me.	4.00	1.31
I would find it easy to get components to do what I want it to do.	3.38	1.51
My interaction with components would be clear and understandable.	4.13	1.55
I would find components to be flexible to interact with.	4.00	1.51
It would be easy for me to become skillful at using components.	3.63	2.07
I would find components easy to use.	4.38	1.69
Perceived usefulness	5.17	1.23
Using components would enable me to accomplish tasks more quickly.	4.63	1.41
Using components would improve my work performance.	5.25	1.16
Using components in my work would increase my productivity.	5.63	1.06
Using components would enhance my effectiveness.	5.25	1.28
Using components would make it easier to do my work.	5.38	1.06
I would find components useful in my work.	4.88	1.46

Diagram locking (n=9)	Average	Std dev.
General		
How confident are you in the ratings that you have made?	5.63	1.06
My usage of diagram locking in my work would be [negative, positive]	4.38	1.19
I predict that I will use diagram locking on a regular basis in the future.	3.38	0.52
Perceived ease of use	5.42	1.18
Learning to use diagram locking would be easy for me.	4.63	1.60
I would find it easy to get diagram locking to do what I want it to do.	5.38	0.74
My interaction with diagram locking would be clear and understandable.	5.88	0.83
I would find diagram locking to be flexible to interact with.	5.38	1.51
It would be easy for me to become skillful at using diagram locking.	5.75	0.89
I would find diagram locking easy to use.	5.50	1.20
Perceived usefulness	3.60	1.32
Using diagram locking would enable me to accomplish tasks more quickly.	3.25	1.04
Using diagram locking would improve my work performance.	3.25	1.67
Using diagram locking in my work would increase my productivity.	3.63	1.06
Using diagram locking would enhance my effectiveness.	3.88	1.46
Using diagram locking would make it easier to do my work.	3.38	1.51
I would find diagram locking useful in my work.	4.25	1.16

5.3 Discussion

5.3.1 Quantitative Experiments

The graphs showing the total, successful and failed number of repositioning requests of the two tests are very different. The graph of the first diagram does not show any failed requests, as the users are all working in their own diagram, and are always the owner of the lock. There are no other users trying to acquire it. The second graph shows an increasing amount of failed requests as more and more users try to acquire the lock at the same time. The total amount of requests also does not increase significantly for 5 or more users. The server is then at max capacity, responding to around 5.750 requests per minute, or 96 per second. As the server starts to respond slower, the lock is held longer and more requests fail, explaining the decreasing amount of successful repositioning requests.

Because the server reaches it's maximum load at around 7500 requests per 120 seconds in the first test and at around 11500 requests per 120 seconds in the second test, the cost of failing a repositioning request can be seen. Failing a repositioning request due to the lock not being acquired is apparently very cheap, as in the second test more requests can be sent within the 120 second window. The user gets a response more quickly, and can then also retry again more quickly.

This can be explained by the differences in database reads and writes between successful and failed repositioning requests. When a request for repositioning is received, the server first reads the state of the lock on the diagram. If it is free or expired, the server will execute a write, writing to the database the current time to reset the lock expiration. If it fails however, due to someone else owning the lock, the server will immediately return a response with an error code. This is much faster than actually handling the request when the lock is free, as that first requires a write to update the lock and then also a read and write to the database to reposition the node. Failing the lock check will only require a single read from the database.

This shows that the locking system is very cheap in terms of server load. The main cost are the database accesses. The test also showed that the server itself when run on consumer level hardware can respond to approximately 5.750 requests per minute when users are working on the same diagram, or 3750 requests per minute when those users are working on separate diagrams where each request is successful. This indicates that the servers should be more than able to support a significant amount of users collaborating and designing diagrams at the same time.

5.3.2 Qualitative Experiments

The answers of the users on the questionnaire are mildly positive for their experience with using components, as overall they rate the impact of components in their work 5.25. The average answer on the usefulness of the components system is 5.17, indicating that users clearly see the value of using components to share their work. The ease of use is slightly rated slightly worse. The worst rated question was whether respondents found it easy to get components to do what they want. Verbal feedback from users was that they thought that more control over component synchronization would give them more agency over their work.

Diagram locking was rated quite high in the ease of use category. Users found it very easy to use and responded that they think they would quickly become skillful at using diagram locking. The only ease of use question that was given an average rating of lower than 5 was the question whether they would find it easy to learn to use. Verbal feedback from respondents was that they thought they would need some time to get used to this system where users need to manually hand over control of the diagram to other users. They were more familiar to systems that allowed multiple users to make edits in real time, as they could for example do in programs like Google Sheets.

The experiment was run with a lock expiration time of 1 minute, but if in any future tests a lock expiration time of 60 minutes is tested, the ngUML project could decide to use those longer lock expiration times. As discussed in section 4.1, that might be a more familiar concurrent editing mode for users. Especially as users then still have the option to hand over the lock control to other users before the lock expires.

Due partially to the usrs unfamiliarity to the locking system as a means of handling concurrent users, the usefulness of the diagram locking system was rated quite poorly. The only question that was given a higher rating than four was 'I would find diagram locking useful in my work'. Presumably because the diagram locking system was considered to be more useful than not enabling concurrent users to edit the same diagram at all. The usefulness of course also suffered from the nature of the locking system, where it does not enable seamless concurrent user collaboration without locks as most users are accustomed to in modern software programs.

6 Conclusions

This thesis set out to implement multi user collaboration in a complex existing code base. This required new functionality to let users both work together simultaneously, but also share their work with others. It achieved this by introducing the concept of components and component libraries. Components were implemented by using non intrusive code design patterns and the usage of new API functions to integrate it into the existing codebase. This approach also had the advantage of being easy to extend for future functionality. The same holds true for the diagram locking system. Because the diagram locking system is required no new protocols such as websockets, or different API functions, it was able to convert all existing API functions in the code base to functions that allow for concurrent users.

The tests that were conducted confirmed that both the component system as well as the diagram locking system satisfied the need to both share work and work together. The approaches that were chosen to implement these concepts did however have their disadvantages for the user experience. The component system does not yet offer enough control to the user and the diagram locking system is obviously not as good as true concurrent editing. The diagram locking system had however no severe impact on the server load, as the extra database accesses that need to be performed to check the status of the diagram lock are very cheap. It also did not need to rely on more expensive protocols such as websockets or long polling. By identifying all the requirements and limitations in the design phase of the project, both systems could be implemented and completed within the scope of this thesis.

7 Future work

As mentioned previously, an important design consideration was the ability to extend the functionality that was implemented in this thesis. There are numerous things that could be added to the component system to enable the user to collaborate with others even more seamlessly. Or even to add components to other areas of the ngUML project, instead of only the diagram phase.

Natural text components Because the component server was designed to be completely agnostic to what is actually in the JSON, you could easily extend this to natural text. Components could be created by copying and pasting parts of your text into a textfield, and then click on a button to create a component. This component would then be sent to the component server as simply a JSON object with a single field called "content". Users could then load the text components back in and copy and paste the relevant parts of the component into their text.

Synchronization strategies More synchronization strategies could be implemented to give more control to users on how the components in the diagrams are updated. A more conservative version could be created that keeps the users changes to nodes. On the other hand a more strict version could also be implemented, one which disconnects any existing edges from the component entities to other non component entities. The implementation of more strategies would also require a new UI element that allows the user to pick the strategy that they would like to use to synchronize a specific component.

Views Component views could allow users to create a layered type of component. Each layer exposes a subset of the component. A layer could be given a name such as ”Simple Customer”, ”Customer - with address” and ”Customer - complete”. Importers of the component could then select the layer of information that they need. By using this layered system, the importer does not have to clutter it’s diagram with unnecessary information, but does not have to edit the component to modify it.

Versioning It would be very useful to users if they could view previous versions of a component. Users could be able to view the history, see who did what and retrieve parts that were once removed but should be introduced back into the component. This would not be very difficult to implement. In the backend, you could add a version field to a component, and simply upload new versions of a component as a new component, but with an incremented version number. Then you only need to create UI to allow a user to pick a certain version of a component.

Application connections You could let the applications that are generated from the UML communicate together. Then a warehouse application could tell a delivery application to deliver a certain product to a certain client. This could be done in real time with the current polling model. The applications would be hosted on a server, while the client simply polls the server every 60 seconds for new or updated actions by other clients, and sends a HTTP request to the server when the client sends an action. Of course, this would require a way to let applications connect together by letting them communicate via an API.

Besides future functionality, more qualitative research could be done to investigate what users need most to enhance collaboration experience. Ideally this would be a larger experiment with more users who will work with the ngUML software for a longer period of time to design a real product. Then the other questions of the TAM questionnaire could be answered by the respondents as well. Their feedback would be invaluable to decide what functionality should be designed, and in what order.

References

[BLFF96] Tim Berners-Lee, Roy Fielding, and Henrik Frystyk. Hypertext transfer protocol–http/1.0, May 1996. https://pages.cs.wisc.edu/~cao/http1.0-rfc1945.html Last accessed on 16/6/23.

[Dav85] Fred D. Davis. *A technology acceptance model for empirically testing new end-user information systems: Theory and results.* PhD thesis, Massachusetts Institute of Technology, 1985.

[DJC] Django channels. *Django Software Foundation.* Last accessed on 16/6/23: https://channels.readthedocs.io/en/stable/.

[FL08] Andrew Forward and Timothy C. Lethbridge. Problems and opportunities for model-centric versus code-centric software development: A survey of software professionals. In *Proceedings of the 2008 International Workshop on Models in Software Engineering*, MiSE '08, page 27–32, New York, NY, USA, 2008. Association for Computing Machinery.

[FM11] Ian Fette and Alexey Melnikov. The websocket protocol, Dec 2011. https://www.rfc-editor.org/rfc/rfc6455.html Last accessed on 15/6/23.

[HBHH] Jonatan Heyman, Carl Byström, Joakim Hamrén, and Hugo Heyman. Locust.io. Last accessed on 13/6/23.

[HC15] Itti Hooda and Rajender Singh Chhillar. Software test process, testing types and techniques. *International Journal of Computer Applications*, 111(13), 2015.

[KH06] William R. King and Jun He. A meta-analysis of the technology acceptance model. *Information & Management*, 43(6):740–755, 2006.

[KRM06] Diwakar Krishnamurthy, Jerome A. Rolia, and Shikharesh Majumdar. A synthetic workload generation technique for stress testing session-based systems. *IEEE Transactions on Software Engineering*, 32(11):868–882, 2006.

[LKL03] Younghwa Lee, Kenneth A. Kozar, and Kai R.T. Larsen. The technology acceptance model: Past, present, and future. *Communications of the Association for information systems*, 12(1):50, 2003.

[LSASW11] Salvatore Loreto, Peter Saint-Andre, Stefano Salsano, and Greg Wilkins. Known issues and best practices for the use of long polling and streaming in bidirectional http. Technical report, 2011. https://www.rfc-editor.org/rfc/rfc6455.html Last accessed on 16/6/23.

[MG15] Nikola Marangunić and Andrina Granić. Technology acceptance model: a literature review from 1986 to 2013. *Universal access in the information society*, 14:81–95, 2015.

[Obj17] Object Management Group. *OMG Unified Modeling Language*, 2.5.1 edition, 5 2017.

[Ozk19] Mert Ozkaya. Are the uml modelling tools powerful enough for practitioners? a literature review. *IET Software*, 13(5):338–354, 2019.

[PN12] Victoria Pimentel and Bradford G. Nickerson. Communicating and displaying real-time data with websocket. *IEEE Internet Computing*, 16(4):45–53, 2012.

[PS19] S. Pradeep and Yogesh K. Sharma. A pragmatic evaluation of stress and performance testing technologies for web based applications. In *2019 Amity International Conference on Artificial Intelligence (AICAI)*, pages 399–403. IEEE, 2019.

[RGSC21] Guus J. Ramackers, Pepijn P. Griffioen, Martijn B.J. Schouten, and Michel R.V. Chaudron. From prose to prototype: Synthesising executable uml models from natural language. In *2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, pages 380–389, 2021.

[WTSW95] Walter Willinger, Murad S. Taqqu, Robert Sherman, and Daniel V. Wilson. Self-similarity through high-variability: Statistical analysis of ethernet lan traffic at the source level. *SIGCOMM Comput. Commun. Rev.*, 25(4):100–113, oct 1995.