# Lessons Learned: Rationale Management Team

Lauren DiCristofaro, Abdulkarim Egal, Vishal Patel

The rationale management team has been working on implementing methods for a user to document the rationale behind the decisions they make about UML diagrams. Given that the ClubUML application handled comparing and promoting diagrams, our team worked on a structured way of inputting and viewing promotion reasoning details. In the near future, we will further extend this functionality to create a decision entity that represents the ground truth diagram. This ground truth diagram essentially represents the current model that the user/user team should be working toward.

## Subject Matter

* Rationale Management is used to capture the motivation behind a decision including the Issues, Alternatives, Criteria, and Argumentation.
* Rationale can be applied to any development decision, so all diagram decisions form a subset of potential decisions that can be rationalized.
* Understanding the history of decisions that led to the current state can be helpful in understanding how to make new decisions.

Rationale management is a hugely important part of software development process, so it formed the basis for 3 newly created use cases that help the user better record and understand the decisions being made throughout the development process. In particular, we wanted to capture a way for the user to structure their rationales so that it is easier to compare rationales and understand reasoning in retrospect. To structure the input, we used prompted the user with established rationale terminology. Relating to the existing project entity, there may be several diagrams that show different ways of representing similar information or design decisions, from these, the user can choose a diagram that they feel is better representative. The user can promote this diagram as a preferred solution. We used the existing compare functionality to establish a solution alternative, which is the compared diagram in this case. The summary represents the unstructured argumentation and discussion in favor of the chosen diagram. The issue and issue relationship identify a particular problem that the diagram may solve and how it solves the problem respectively. The criteria and criteria relationship identify what requirements or desired properties one diagram may have and what about the diagram better addresses those requirements respectively. Because this information is specifically prompted for, a user is more likely to clearly state their reasoning in a way that can be understood more readily by both person and machine.

There are a wide variety of decisions that can be made during development, but for a team very focused on process, many of these decisions will be captured in UML diagrams. These diagrams are specifically meant to clearly capture solutions, so the diagram evaluation process is an excellent place to clearly state rationale and formalize the decision making process. This decision process may be used for any of a number of different potential diagrams that could by uploaded to the ClubUML system. Whether the team is deciding which use case diagram best addresses customer priorities or which sequence diagram best represents how system processes should interact with each other, an authorized team member can use team member rationales to resolve the question to a set decision.

Recording the rationale behind decisions becomes especially important when making decisions in the future about subsequent or related issues. Understanding why the initial decision was made provides resources for improving future decisions and revising previous decisions. Seeing the reasoning behind the path to the current ground truth helps a decision maker to see the direction the project is headed and better understand the perceived project perspective. Even with the current ClubUML project, we can see how understanding the decisions of previous classes might indicate the most productive decisions for future growth. As such, we created a use case for tracking the history of decisions made and providing an intuitive way to view them together. A privileged user could create a decision and aggregate the most relevant rationale for why this decision seemed the best at the time. In the future, they might go back to select an alternative given new information, and it would be easy to look up the alternatives. Or they might want to make a decision about a library but decide that it is not feasible because it’s related to a previous library that was reject for its incompatibility. This provides powerful insight into the current project and how it should evolve.

## Implementation

* Requirements, design, and testing documents provide good coverage throughout the development process for understanding how new functionalities fit in with the existing codebase.
* Expect time to learn about the codebase, new technologies, and how they all connect with each other.

Our team worked with a few different document types throughout the development process that really facilitated development and a good understanding of the value that we created. Use case documents were established early to clarify the value of the new functionality and record a user’s perspective of what we would be building. Our team created a couple of different types of design documents throughout different parts of the software development process. We started with UI documents and mockups as a quick way to establish the basics of what we were creating. These often brought up questions that fleshed out design basics and clarified details from the use case. We created sequence diagrams and class diagrams at different points in the development process, and we found that creating them earlier helped to clarify more technical issues earlier so that no one ended up confused or doing the wrong work. Once we had done basic local testing, testing documentation clarified validation criteria and provided a method for integrating into the baseline.

One of our largest technical challenges came from understanding and working with technologies that we’re less familiar with. It is important to establish familiarity levels with different technologies early in the development process to better understand the difficulty levels of certain tasks. For the rationale team, we faced some challenges with better understanding javascript and what role it played in the project as a whole. Javascript was closely woven throughout the existing project, so the team had to learn more about the formatting and syntax of javascript, but also about how it interacts with html, css, and the java enterprise project via jsps. All of these components were a key part of the general flow of the application. Understanding this flow was essential for integrating new functionalities into the existing application structure, but it also might have been better facilitated with a more intuitively organized codebase such as a centralized css file.

A learning curve also came up for a lot of the class in the use of git technologies. Not many class members were familiar with git, and this presented challenges with sharing development work between or even within the smaller task teams. By quickly addressing this knowledge gap, we might have had better success with integrating work and mitigating development risks.

Process

* Lightweight processes help to speed development time, but this comes with communication and documentation caveats.
* Agile techniques and collaboration were essential to creating good work products.
* Basic design documentation should be created before developing, but documentation update time should also be specifically allotted and built into acceptance criteria.

Software lifecycle processes discussed in the Software Engineering class tend to be very formalized which is good for eliminating ambiguity, but it does tend to make these processes more heavy weight. The processes gave us a good framework for exploring possibilities, but we only followed them very loosely. The disadvantage of this is that there’s greater potential for communication issues and deviation from the needs of the project, but we saw great advantages in our level of productivity because we weren’t as consumed with the meta-details of every decision. We think that we still mitigated risk well by regularly demonstrating functionality. Another disadvantage is that we may end up leaving less documentation for future maintenance teams, but we’ve found that having too much documentation can sometimes be a burden in itself. The main advantage in maintenance documentation comes from concise well-written documents meant to directly inform and aid future developers.

We have found that agile development fits well with our development requirements for this semester long project. Agile has made the schedule very clear, and showing deliverables early ensures that we’re making good progress throughout the semester. The semester is a relatively short period of time compared to most larger projects, so it was important to get organized as quickly as possible. Our agile development processes had well-defined organization already built into it. Roles were very well defined, and it was easier to keep track of task commitments. Agile also encouraged a progressive development style that took us through logically stops and helped us break up work into manageable and evaluable tasks. This alongside meetings and other communication techniques helped to bring a clear picture of what needed to be done and identify miscommunication issues relatively early. Good communication techniques were especially important, and setting up further team meeting encouraged a more collaborative development environment. By increasing the number of meetings outside of class, we found that we could share expertise, integrate work and clarify goals more easily. We also used these meetings for informal document and code reviews that helped to address many small issues that might have otherwise been missed.

We’ve seen from our own work and that of past classes that it’s difficult to keep documentation up to date, and it often doesn’t represent the existing codebase. But if written correctly, documentation may also greatly improve understanding of the project and its goals. We think it would have been helpful to push more design documentation ahead of implementation, but with that, it’s also important to remember to update documentation afterwards and clearly specify what is current functionality versus planned or legacy design. The coding process provides a better context for the decisions that will create a successful outcome, and often when these decisions are made at implementation time, the design documents may be forgotten and not updated to reflect the new decisions. Part of the tasking involved in creating some functionality should include time for documentation updates so the necessary and appropriate documentation is an integral part of the development process. This should be viewed as a critical priority rather than an afterthought.

## Tools and Technologies

* UMLLab and Visio are UML diagram building software tools that provide an easier user interface than papyrus.
* Firefox and Chrome debuggers for jsps and eclipse debugger for Java provide helpful functionality for local testing and debugging

Papyrus is the UML software used by the previous class. One of the main advantages of papyrus is that it is directly integrated into the eclipse development environment, but this can also cause some issues. Papyrus has some compatibility issues with certain versions of eclipse (Luna was too new for it), so it may require a developer to download a different version of eclipse to just to use it. Papyrus’s user interface is sometimes very difficult to work with because all of it’s functionality is tailored to fit into the eclipse framework which is not necessarily designed for diagram building. There are no easy copy paste functionalities, and typical actions required multiple steps or a lot of searching to find the correct functionality.

UMLLab is an application that’s separate from eclipse, but it has a plugin that allows you to link it to the eclipse workspace. Using this, you can view your code side by side with the diagram that you’re creating which gives the developer a better reference. In general, the functionality is much more intuitively organized and greatly improves productivity when creating UML diagrams. The main downside to UMLLab is that it’s not free, but you can easily get a short-term license, and educational licenses might potentially be available pending further investigation.

Visio is another UML tool that is very easy to use, and there is some advantage to gaining experience with the tool because it is widely used in industry. It is not free to the general public, but it is free to Northeastern students through COE. Microsoft Visio is less compatible with Mac, but there are tools available to read Visio on mac.

Debugging tools have been a key part of the development process. During development, we can use immediate feedback to make rapid changes and adjust functionality on the fly. This has been especially important in the relatively complex codebase that we came into as it may be difficult to get a comprehensive view of the entire application. By using debugging tools, we can get a more localized perspective on the code of particular interest and do quick trial and error assessments of code that we are uncertain of. Chrome and Firefox browsers both come with robust debugging tools for understanding the components of the html pages and javascript code being executed in the browser. The eclipse debugger provides a similar functionality for the java classes by allowing a developer to step through and inspect the instantaneous values of variables.