**Capstone Project Cover Sheet**



|  |  |
| --- | --- |
| **Capstone Project Name:** | **UML Flow Program** |
| **Student Name:** | **Kevin Fode** |
| **Degree Program:** | **BS, Software Development** |
| **Student Mentor Name:** | **Nancy Corea** |

**Table of Contents**

[Introduction 1](#_Toc488397392)

[Review of Other Work 2](#_Toc488397393)

[Rationale 3](#_Toc488397394)

[Systems Analysis and Methodology 6](#_Toc488397395)

[Goals and Objectives 8](#_Toc488397396)

[Project Timeline 11](#_Toc488397397)

[Project Development 13](#_Toc488397398)

[Evidence of Project Deliverables 18](#_Toc488397399)

[Edit Mode Deliverable 18](#_Toc488397400)

[Value Storage Deliverable 19](#_Toc488397401)

[Value Editing Deliverable 20](#_Toc488397402)

[UML Path Deliverable 21](#_Toc488397403)

[Decision Trees Deliverable 23](#_Toc488397404)

[Value Deletion Deliverable 29](#_Toc488397405)

[Stepping Mode Deliverable 31](#_Toc488397406)

[Save/Load UML Deliverable 32](#_Toc488397407)

[References 35](#_Toc488397408)

# Introduction

The purpose of the capstone project was to create a program which could be used to generate a text based version of a Unified Modeling Language (UML) diagram. A traditional UML diagram acts as a visual flowchart. It is used primarily in the planning process, but can also be used during quality control processes to verify that the code written adheres to the proposed project’s design.

Unfortunately, this type of diagram is hard to use for quality control purposes. With a complex program, a visual UML diagram can easily appear as a jumbled mess. The capstone project attempts to alleviate this issue.

The capstone program is a text-based, UML diagram builder. The user is able to enter steps into the program. Each step can be edited, deleted, and reordered. The user is also able to add decision trees which affect the step order. Once the UML diagram is complete, the user can walk through the process one step at a time. The user is responsible for advancing to the next step by hitting a key. This way, it is not possible for the programmer to lose their place in the diagram. In addition, by displaying each step one at a time, the overall layout of the process is hidden from the user, allowing them to focus on the content of each step. In this way, the complexity of the program’s workflow can be more easily digested.

Overall, the creation of the capstone program progressed well. The planning document written for this project helped map out all the goals and objectives that needed to be reached for this project to be completed. The planning process saw the creation of a class diagram for the capstone program. This resulted in the program’s classes being properly thought out and encapsulated. This is the primary reason the project was completed ahead of schedule even though production started a day later than anticipated. The only stumbling block was during the creation of a large section of navigation code. This block resulted in a couple hours of troubleshooting, which probably could have been avoided using different practices. There were a few changes implemented after programming was under way. Once the objectives were completed and all changes implemented, the program was considered complete and the project was brought to a successful close.

# Review of Other Work

The goal of the capstone project was to provide a method to easily validate the programmer’s code and to verify that the code performs the way it was intended. The more that a team commits to the UML process, the more detailed the output can become. However, as the models become more complex, they also become far more difficult to read (Drusinsky, 2006, p. 44). To make the UML diagram easier to understand, the capstone program hides the complexity of the diagram by only presenting the piece the user is currently working on.

This program was produced using the small project management (SPM) methodology outlined by Sandra Rowe (Rowe, 2015, p. 48-56). This process is similar to the waterfall methodology, as it is a linear process designed for smaller projects. SPM uses the same basic life cycle defined in the PMBOK® guide, except that it combines the Executing and the Monitoring and Controlling phase into a single process (Rowe, 2015, p. 53).

The technical writing project document started the initiating process. During this process, objectives were set. These objectives were each assigned a deliverable, which would have to be implemented for the objective to be reached. This process was complete once that document was approved.

The planning process broke down the project objectives into manageable pieces. Class diagrams were drawn up and checklists were created which detailed what was to be done in each component of the final product. While the technical writing project document laid out a schedule allowing for an eight day project, the entire schedule ended up only taking three days. This is due in large part to decisions made during the planning process. The classes of the capstone program were designed in such a way, that each class handles its own processes. Each class has its required accessor methods. The accessor methods allow all classes to have private variables (Levenick, 2006, p.378). This put the onus on the class to have well thought out methods for modifying the contents on those variables. This meant that the classes formed were highly decoupled. Changes to how a class modified its variables were irrelevant to the classes executing those methods, so long as the result remained the same.

The controlling phase used the checklists creating during the planning phase to write the code and to check whether the component met its specifications. This was an iterative process which was restarted every time one objective was met, and another began. There was some difficulty in one of these iterations, due to some bug-ridden code. Because of this, a few hours of troubleshooting were needed. This time could have been better spent had more robust system testing been used (Campbell & Hinds, 1995). These issues are described in later sections regarding the individual deliverables.

The closing process finished the development of the program. The program was delivered to the end user and accepted. Closing documents, such as this document, were developed at that time.

# Rationale

The purpose of the capstone project was to assist the programmer’s understanding of the desired functionality of the program they were attempting to create. This way, the programmer could spend less time trying to grasp the overall complexity of what they were trying to write, and focus solely on the task before them. For the purposes of the program they were trying to code, any information that was not the current focus of the programmer was considered noise.

To accomplish this, the capstone program displays information step by step, and only proceeds at the bidding of the user. The program moves forward in a linear fashion from beginning to end, while allowing the user to answer multiple-choice questions along the way that may result in iterative processes. The user was expected to enter information into the program that would then become the step by step process that could be used for testing and troubleshooting. In this way, the capstone program has a great deal of flexibility in how it can be applied.

As an example, a user could decide to use this program to verify the completeness of a final program. To do this, the user would enter the contents of a UML diagram created during the planning process. They would then use the program to verify the final result by stepping through the program as the result is tested simultaneously. This would be considered to be at the complete picture level.

While the capstone program can be used to test an entire product, it can also be used to build UML diagrams for components as well. This level is considered to be the class or component level of detail. UML diagrams created during the planning process often are not broken down to the point where the inner workings of components are revealed. This is because doing so would make the diagram overly complicated, and would also waste the time of any programmer not involved in the creation of the component in question. However, there is nothing stopping the individual programmer from creating a UML diagram specifically for the component they are working on. In fact, this process is recommended whenever additional understanding is required. After the programmer creates the UML using the capstone program, they can then use the program to test the component they coded.

For a more granular level of detail, the programmer can use the capstone program to analyze the workings of a process line by line. This would be considered the method or process level of detail. The UML necessary for this level of detail would be highly specific, and most likely only used while troubleshooting. This level would include what the process needs to do every step of the way to return the proper result.

Due to the varying levels at which the capstone program can be applied, the feasibility of its applications also vary. At the complete picture level, the user inputting the UML for testing has a choice of either being very general with each step, or spending a great deal of time entering more specific information. This will depend on the size of the finished result. The larger the result being tested, the less feasible it is that the capstone program will provide value.

However, at the component level, the capstone program has a much higher chance of adding value. At this level, the programmer can start their task by creating the UML diagram with the capstone program. This will provide a greater understanding of the problem and allow the user to focus on each step as they come by them using the capstone program.

At the method level, the capstone program is once again likely to have little value. Because this level requires such a precise level of detail, it is highly unlikely that a programmer would want to spend the required amount of time to implement the capstone program at this level. If they did choose to, they would probably only do so if they were completely stuck. Because the UML required at this level is so detailed, this process would most likely result in a half finished UML since the programmer would probably stop as soon as they happened upon the step they were missing. It is debatable as to whether it would be more time efficient for the programmer to try to use the capstone program for this level of troubleshooting, or simply start over.

For these reasons, the primary purpose of the capstone program is to support programmers who are operating at the component level of testing and troubleshooting. Some users may find it useful at the other two levels, but this was not the driving factor for building the capstone program.

# Systems Analysis and Methodology

Code validation is a key component of the software development process. The programmer can choose to do this in a number of ways. One way would involve tracing through a program’s UML diagram with a finger while simultaneously executing the program. Another way the programmer could try to validate their code would be to work off a whiteboard, making a checkmark at each step. Both of these approaches can be problematic. It is easy with both approaches for the programmer to lose their place and become overwhelmed.

To alleviate these issues, the capstone program was made responsible for keeping track of where in the process the programmer currently is. No changes to the UML diagram are made as the user steps through it, maintaining the diagrams integrity. Any interruptions to this process are irrelevant. The programmer can simply pick up where they left off without having to worry about where in the code they were before they stopped. The capstone program keeps track of their location in the UML diagram for them.

As discussed before, the methodology used for the creation of this program was the small project management (SPM) methodology outlined by Sandra Rowe (Rowe, 2015, p. 48-56). This methodology is made for small projects like the capstone project. It is also made for projects with small teams.

The initiating process began with the creation of technical writing project document. This document identified the stakeholders for this project as any programmer wishing to make their validation and troubleshooting processes easier. These programmers would also have to be willing to invest the time it takes to load the UML diagram into the proposed program. The people who employ these programmers were considered secondary stakeholders. The initiating process was completed when this document was approved.

The planning process started with the collection of requirements, which were used to create a scope statement. This project’s scope statement was the following:

To create a program a programmer could use to build a basic Unified Markup Language (UML) diagram programmatically. This program will include the ability to create steps that contain the step’s text and links to other steps. Whenever multiple next steps can be chosen, text describing the different choices will be displayed in a menu which asks the user how to proceed. The user will be able to create an unlimited number of steps. The user will also be allowed to order these steps as they see fit, including iteratively. This program will also have the capability to save and load UML diagrams to the hard disk. This project will be completed in seven working days.

Another part of the planning process was the creation of a work breakdown schedule (WBS). Objectives, and the deliverables that met those objectives, were set during this process. Goals, objectives, deliverables, and schedules were all set during this phase and are discussed in later sections of this document. A class structure for the capstone program was also developed during this process. The final class diagram is provided in a later section of this report.

The planning process for SPM also included some activities that were not completed during this project’s development. Costs, risks, and communication plans were considered irrelevant for this project. The only cost considered for this project was time. As for risks, it was considered that any risk would be dealt with by the only programmer on the team. No risks could be mitigated for this project. There was also no communication plan considered, since the only stakeholder and the programmer were the same person. As this project is now completed, it can be reported that these assumptions were proven correct.

The controlling process was where the programming began. During the final phase of the planning process, checklists were created of all individual requirements of each deliverable. These checklists are provided later in this document. While there are a number of activities in the controlling process of SPM, the only two used during the capstone project’s development involved the reviewing and approval of deliverables and the managing of any scope changes needed (Rowe, 2015, p. 117). The rest of the activities were considered redundant for a one person project team. While there was a change process in place for this project, it was used sparingly during its development. These changes are described in a later section.

The final process in SPM is the closing process (Rowe, 2015, p. 129). This process began once the project was considered complete, by meeting all the requirements set forth in previous processes. This project closure report was generated to document lessons learned while working on the project. The project will be considered officially closed once this document is accepted.

# Goals and Objectives

The only goal of the capstone program was to provide a method to easily validate the programmer’s code and to verify that the code performed the way it was intended. The capstone program provides an alternative traditional validation methods, insuring the accuracy and timeliness of the results. The capstone program achieves this goal by accomplishing several objectives:

* the program has an edit mode which allows the user to create and change a UML flow chart
* the program has a stepping mode to allow the user to logically flow through the UML flow chart once it is complete
* the program has a method for storing entered values for future use
* the program has a way for the user to edit values that were previously entered
* the program links values together in a logical string
* the program allows the user to create multiple paths to different following values
* the program allows the user to delete values
* the program allows the user to save and load data to and from a file

The first objective was that the program had to contain an edit mode. This mode allows the user to freely create and edit a UML flow chart. Unless the user has previously saved UML data, the user would always use this mode first. This objective was met early on in the project. Originally, this mode was only some dummy text. The edit mode evolved throughout the project as each objective was met.

The second objective was that the capstone program needed a stepping mode. This mode is used to step through the UML flow chart created in the edit mode. This is the process the user executes to verify the functionality of their code. This objective was also met early in the project. Like the edit mode objective, the stepping mode was only some dummy text when this objective was met. It was not until the completion of the sixth objective, which allows for multiple paths, that this mode was finalized.

The third objective was that the capstone program had to be able to take user entered values and store them for later use. When the programmer later executes the stepping mode of the program, these values are sent back to the programmer. These values are used to let the programmer know what step is currently being executed and what tasks should currently be under way. This objective was easily reached through the creation of getter and setter methods.

The fourth objective was that the capstone program had to allow the user to edit entered values should the need arise. This is another function of the edit mode. Without this, the programmer would be required to enter values correctly on the first try. This objective was reached at near the same time as the previous objective. It was only necessary to create the menu options which would allow the user to do so.

The fifth objective was that the capstone program had to link these values together in a logical string. This string is created and edited within the edit mode. This string tells the stepping mode which value will be the next for display. Completing this objective started the meat of the overall project. This is when the overall logic of the linked lists that serve as the basis for the program’s functionality was developed.

The sixth objective was to allow for multiple paths from the current value to different following values. These junctures allow the user to make multiple choice decisions as to which step to continue with. This objective was key. It is this decision path process that creates the greatest confusion while trying to troubleshoot and while trying to validate a process. Without achieving this objective, the programmer is left with a strictly linear process, which is not difficult to model or test. It was therefore imperative that the capstone program meet this objective. Once this objective was completed, the stepping mode had reached its final state.

The seventh objective was to allow the user to delete values if necessary. Creating a UML diagram can be a messy process. With the capstone program, the user is not forced to start over if a value entered previously turns out to not actually be part of the process being mapped. Once this objective was reached, the edit mode was in its final state.

The eighth objective was to create the ability to save and load the UML created during the edit mode process. Because the UML necessary to map a process can be quite large, the capstone program has the capability to store this information for later retrieval. Once this objective was reached, the capstone program was complete.

# Project Timeline

The projected timeline for this project is laid out in the table below. Since the first four objectives were considered simple and interrelated, it was expected to be completed on the first day. Allowances were made for weekends. Due to outside forces, the project started a day late. However, once the project started, it became clear that the durations set during the planning were heavily padded.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Project Deliverable or Milestone** | **Planned Duration** | **Planned Start** | **Actual Start** | **Actual Duration** |
| Edit Mode | 1 Hour | 7/13/17 | 7/14/17 | 15 Min. |
| Stepping Mode | 1 Hour | 7/13/17 | 7/17/17 | 1 Hour |
| Value Storage | 3 Hours | 7/13/17 | 7/14/17 | 15 Min. |
| Value Editing | 3 Hours | 7/13/17 | 7/14/17 | 15 Min. |
| UML Path | 8 Hours | 7/14/17 | 7/14/17 | 2 Hours |
| Decision Trees | 8 Hours | 7/17/17 | 7/14/17 | 3 Hours |
| Value Deletion | 8 Hours | 7/18/17 | 7/17/17 | 1 Hour |
| Save/Load UML | 24 Hours | 7/19/17 | 7/17/17 | 4 Hours |

The edit mode, value storage, and value editing objectives ended up being much easier to meet than originally expected. This is due to a misunderstanding of what each of these objectives would truly mean once the project was underway. As stated previously, both the edit mode contained only dummy text originally. It existed programmatically when their milestone was reached, but it did not contain any of the desired functionality until much later in development. The projected durations had accounted for much more functional code in these areas.

The UML path and decision trees objectives also took finished ahead of schedule, but in retrospect, these objectives should have taken even less time. Since these sections of code contained a lot of different logical paths, it took some time to code them. Instead of testing the code periodically, it was decided to delay testing until all the code was written. This resulted in several nested errors that took some time to troubleshoot. This could have been avoided with more progressive testing techniques (Campbell & Hinds, 1995).

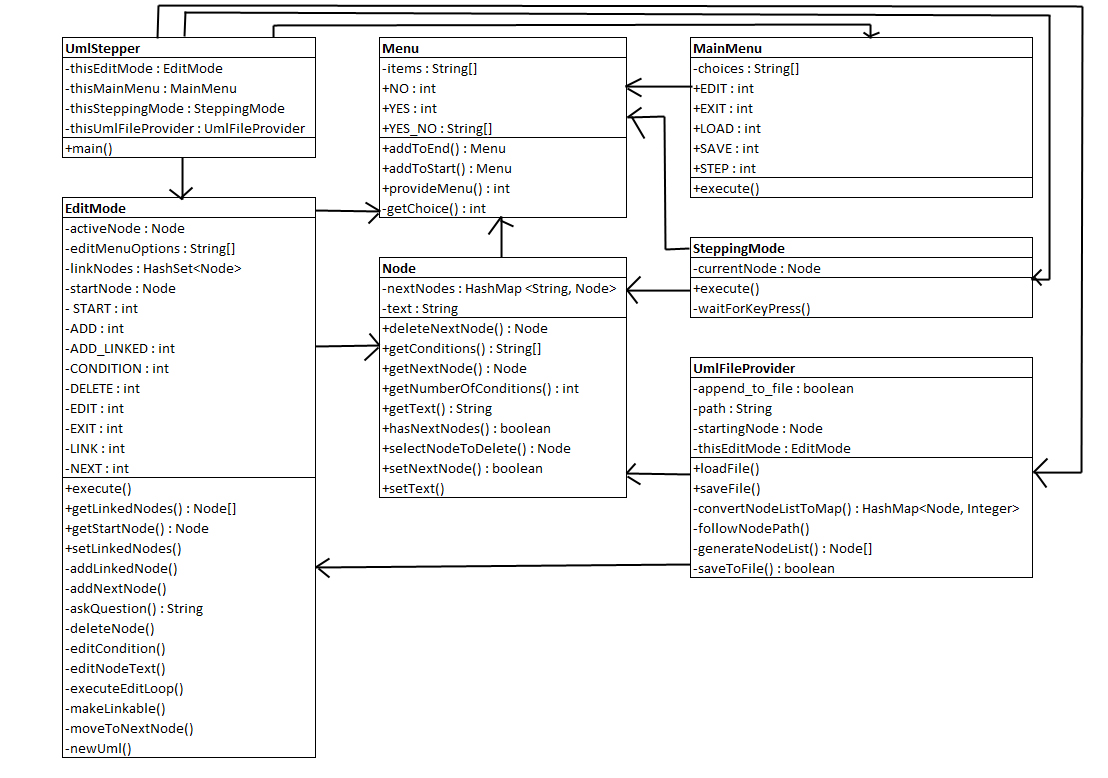
The value deletion objective was something else that ended up taking far less time than originally anticipated. The projected duration of that task had to do with the idea of doing rigorous testing to prevent null values from propagating throughout the node paths. As it turns out, the solution used had nothing to do with nulling values. Testing was far smoother as a result.

Originally, the stepping mode was listed as the second objective to be reached. However, upon implementation it quickly became clear that this objective would be impossible to complete until after several other objectives were met. The original stepping mode simply printed some dummy text. It wasn’t until after the node paths were fully implemented that this objective was reached.

The save/load UML objective was the final objective which finished ahead of the schedule. This objective’s duration was intentionally padded. This was done for two reasons. For one, it had been a while since the team had dealt with programming that created text files. The second reason this was done was to account for any changes in availability of the project team member.

# Project Development

During the planning phase, it was important to not only to list objectives, but to also provide a good framework for achieving them. To make sure that the project adhered to proper object oriented design, a class diagram was created. The final class diagram is in the figure below.



While most of the classes map to several objectives, the Menu and MainMenu classes do not map to any objective. Since this program is largely menu driven, and erroneous user input is a major cause of bugs, it was decided during the planning phase to make the creation of and returns from menus the responsibility of a single class. This decision contributed to the success of the overall program.

The capstone program had a number of deliverables to mark the progress of the project. Each deliverable was verified using a checklist created during the planning process. The project deliverables are in the table below.

| **Project Deliverable** | **Objective** |
| --- | --- |
| Edit Mode | The program must have an edit mode to allow the user to create and change a UML flow chart. |
| Value Storage | The program must have a method for storing entered values for future use. |
| Value Editing | The program must have a way for the user to edit values that were previously entered. |
| UML Path | The program must link values together in a logical string. |
| Decision Trees | The program must allow the user to create multiple paths to different following values. |
| Value Deletion | The program must allow the user to delete values. |
| Stepping Mode | The program must have a stepping mode to allow the user to logically flow through the UML flow chart once it is complete. |
| Save/Load UML | The program must allow the user to save/load data from a file. |

The Edit Mode deliverable marked the completion of the bare bones menu system for the proposed project. This mode is similar to the stepping mode, in that it steps through any created values, called nodes, in a similar fashion. This mode uses that stepping feature to keep track of which node is currently the active node.

A few changes were made to the original plan upon implementation. First, since the original plan only allows the user to navigate downstream, another menu option was added to allow the user to change the current node to the starting node. This allows the user to move back to the start without exiting the edit mode.

The second change that was implemented was changing the “delete node” option to “delete next node.” It was decided that allowing the user to delete the node they were currently in was problematic. Besides, the user needed to have the ability to unlink nodes from each other in case an error was made. This solution kills two birds with one stone.

The third and final change implemented was to add a “Make Node Linkable” menu option. This way, when the user chooses to link to an existing node, they are presented with a small list of nodes they have marked linkable. The original plan was to have a potentially much larger list of all nodes.

Another change that was considered was adding a menu option to bookmark nodes. The idea was to give the user the ability to freely navigate to a node other than the starting node. This change was ultimately rejected as it was considered too robust for the task at hand. This change may be implemented in a future version of the capstone program.

The Value Storage deliverable involved the planning and development of the data object, which is used to store the values entered by the user. Each value is stored in its own data object. When the data object is created, a default value is stored in its text attribute.

In the original plan, these data objects would contain unique identifiers to be used by later deliverables for linking existing nodes. The idea was that when a user wished to link to an existing node, they would be presented with a list of all nodes. During implementation, this idea was discarded. The reason for this is because it was thought that with large UML diagrams the complete list of all nodes would be overwhelming. The idea of making a node linkable was implemented instead. This way, the identifier used for each node is the node’s reference, which is already unique.

The Value Editing deliverable added the additional functions to the edit mode of the program. These functions allowed the user to change the value stored in the data object. The edit mode only allows changes to be made to the active node.

The UML Path deliverable added additional information to the data object. Each data object stores a reference to the next data object in the chain. Additional coding was added to the edit mode to allow the user to navigate to the next node. For this deliverable, the movement between nodes is strictly linear.

This is the point where the project started to ran off the rails a bit. Because these nodes would contain multiple possibilities in the future, a HashMap was used instead a simple Node reference. Since there did not seem to be any need for storing the keys for this map, several operations were needed to get the values out of the HashMap. With all the different operations necessary to pull these values without keys, a lot of code was written before any testing occurred. The result was a nested cluster of errors that took some time to troubleshoot. To overcome this, several print statements were added to list the value of objects as they navigated through the different conversions. It took some time, but the code was ultimately fixed.

The Decision Trees deliverable added the ability to make multiple choice decisions as to which node to navigate to next. The edit mode was modified to present a multiple choice menu whenever a decision tree was detected. This was also the point where linkable node logic was implemented.

For this deliverable, the only change to the original plan involved allowing the user to make a node linkable, and giving the user a list of those nodes when they request to link an existing node. As stated previously, it seemed to be a better idea to try to limit the length of this list by only including nodes the user has chosen. In retrospect, it probably would have been a good idea to include the ability to undo making a node linkable as well.

The Value Deletion deliverable added the ability to delete a node from the current node’s next node list. Originally, the idea was to allow for the deletion of the current node. This idea was changed when it became clear that there needed to be some way for the user to delete a link to a next node. It is important to note that the methods created for this purpose do not actually delete nodes; they simply dereference them. This distinction is important, since this approach makes it impossible to create null pointer exception issues.

The Stepping Mode deliverable added a simplified version of the edit mode. The only options available in this node is to proceed to the next node. When more than one node can be navigated to, the user is presented with a menu with a list of conditions which lead to the next nodes in the chain.

During the implementation of this deliverable, it was quickly discovered that this objective was written out of order. Instead of being the second deliverable, this objective should have been the seventh. As such, the original code had no functionality whatsoever. It was not until much later that this mode was fully implemented and this objective reached.

The Save/Load UML deliverable added the ability to save and load previously entered node information. This information is stored in a simple text file. It can then be retrieved using another menu action.

Once this last deliverable was completed, the controlling process was complete. After the last bit of programming was complete, it became clear that this more a beginning of a project idea rather than a full version. There are other things that could be implemented into this project were not. One of these things would be allowing for the bookmarking of nodes, so that a user could more easily jump between different areas of the UML while editing. Another idea would be to allow for the use of persistent data objects that could track the values of user defined variables as the user runs through the code.

# Evidence of Project Deliverables

## Edit Mode Deliverable

The checklist for the Edit Mode deliverable was as follows:

* Lists the current step
* List conditions for next nodes (if more than one)
* Lists menu options for the following:
  + edit the current node
  + go to next node
  + add next node (creates a new node)
  + add an existing next node (links this node to an already existing node)
  + edit a condition (for decision trees)
  + delete node
  + exit edit mode

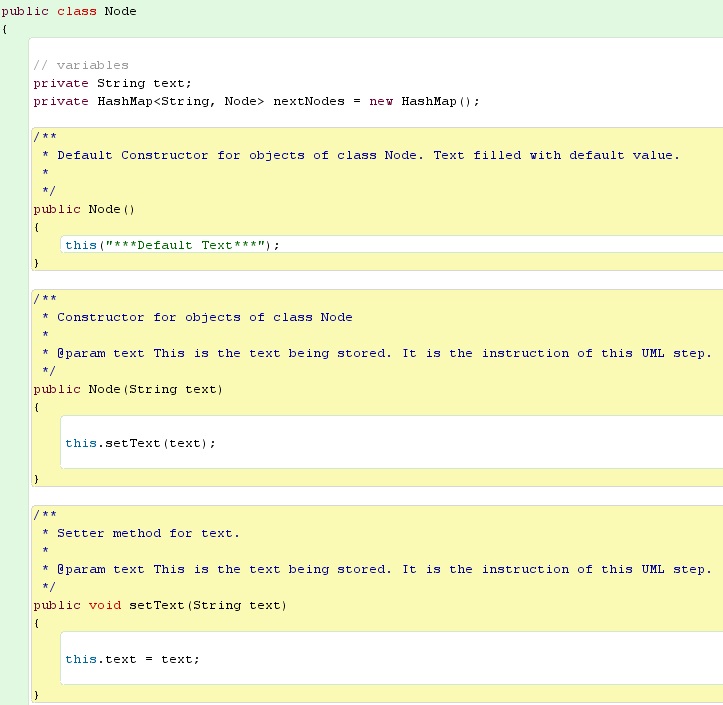
As the figure below shows, this deliverable ended up being a simple menu. The menu for the entire project is seen at the top. Before the menu, the current step and conditions for the next node print. The figure below does not have text for conditions, but an example further down does. The menu has options for all the checklist items plus the ability to make nodes linkable and to return to the start of the UML.



## Value Storage Deliverable

The Value Storage deliverable only involves code which is shown in the figure below. It is a data object which stores values for future use. It had the following checklist:

* create a data storage object (node) which can hold user data
* create a unique identifier for each node (this item was not implemented)

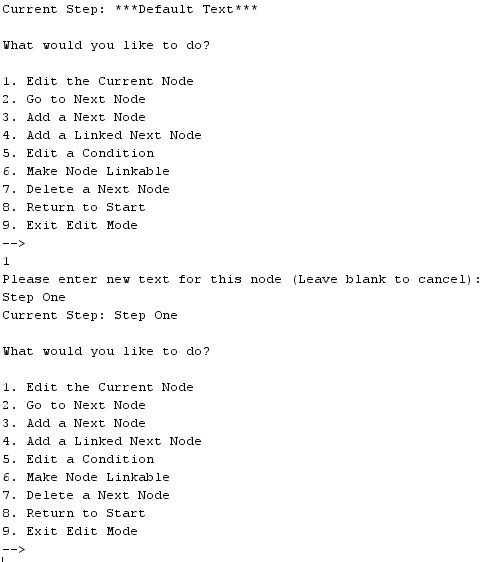


## Value Editing Deliverable

The checklist for the Value Editing deliverable was as follows:

* take user input to change a node’s text
* swap out the node’s text with the inputted text

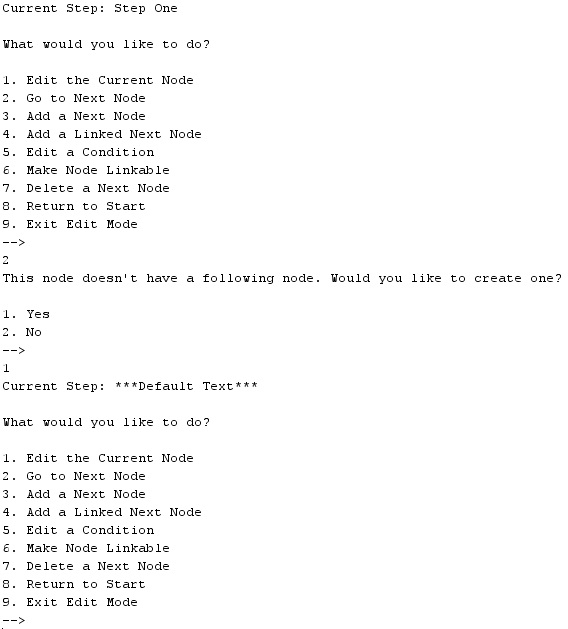
As shown in the figure below, this operation is a simple use of getter and setter methods. The user is prompted for a new value, and the program stores it.



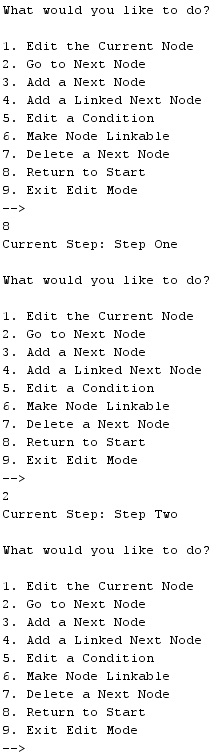
## UML Path Deliverable

The checklist for the UML Path deliverable was as follows:

* add a reference to a node for the next node in the chain
* allow the user to navigate to the next node



In the example above, the original node does not have a next node. The user is prompted if they would like to create one. In the example below, the current node has a next node. In the second case, the program simply navigates to the next node.

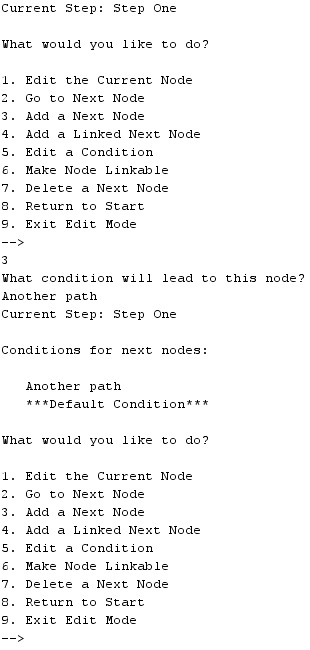


## Decision Trees Deliverable

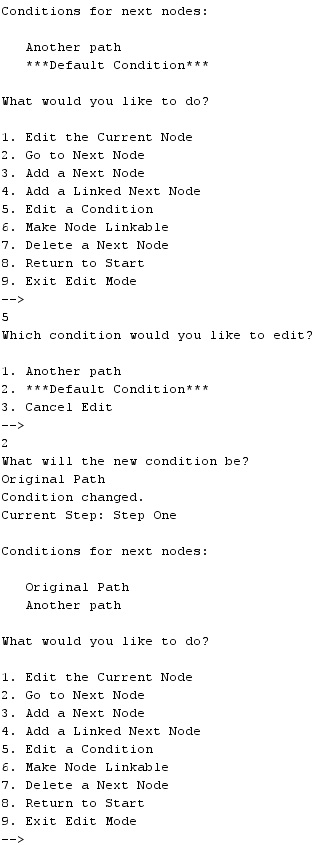
The checklist for the Decision Trees deliverable was as follows:

* Allow user to add more than one next node
* When more than one next node exists, prompt the user with conditional text when switching nodes
* when more than one next node exists, allow the user to edit conditional text
* allow the user to link to an existing node

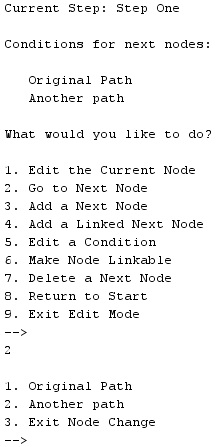
In the following figure, Step One already has a next node. The user is prompted for the condition that will lead to this second path. Now that Step One has two next nodes, a list of conditions appears below the current step.



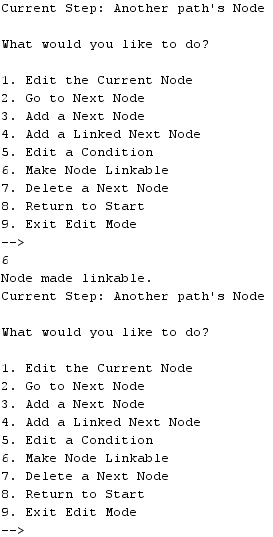
Now that Step One has multiple conditions, it is now possible to edit the conditions that lead to those nodes. The figure below shows the user changing the text of the default condition.



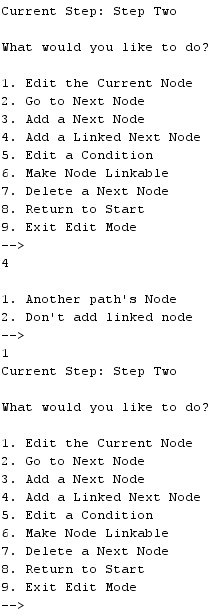
Since there are now multiple next nodes, the program will now ask the user about which node they wish to navigate to when selecting “Go to Next Node.” This is shown in the figure below.



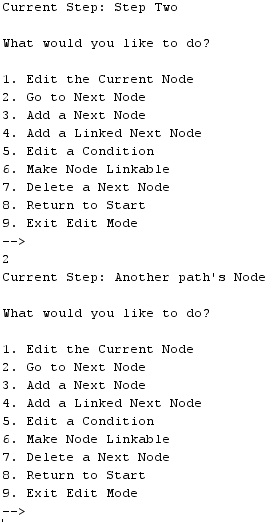
The last task for this deliverable was to allow the user to link to an existing node. The first step in this process is to make a node linkable. This is shown in the figure below.



Now that this node is linkable, the user is able to navigate to another node and link the existing node. This is shown in the figure below.



Now that the user has linked this node to the existing node, they can navigate directly to the node they linked. This is shown in the figure below.

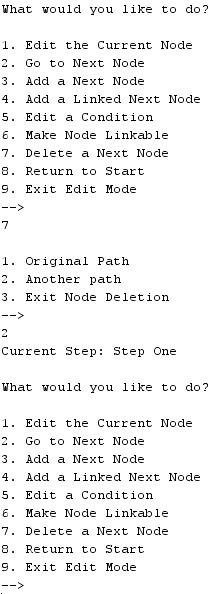


## Value Deletion Deliverable

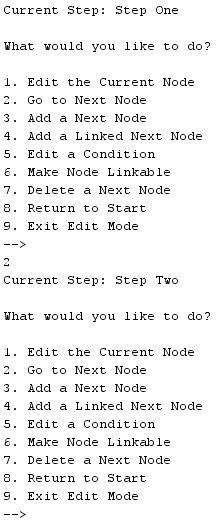
The checklist for the Value Deletion deliverable was as follows:

* Allow the user to remove a node from the node chain

In the figure below, the user deletes one of the next nodes. Since the current node no longer has two next nodes, the program no longer prints a list of conditions.



To demonstrate that this node is no longer linked to, the figure below shows the user navigating to the only next node available.

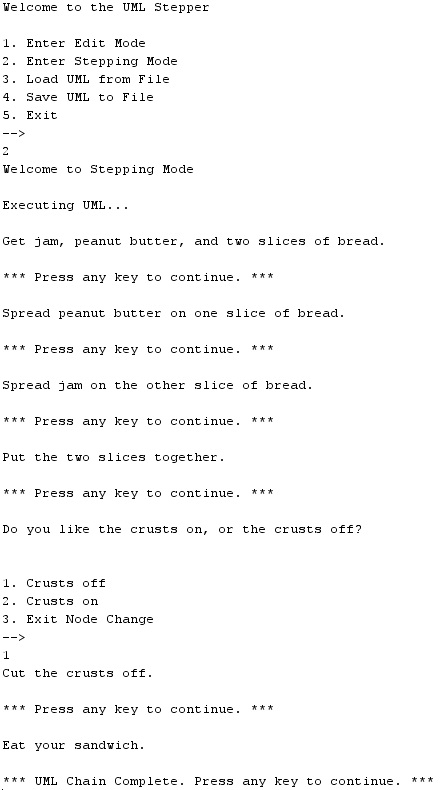


## Stepping Mode Deliverable

The checklist for the Stepping Mode deliverable was as follows:

* Lists the current step
* If there is no next node, alert the user that the stepping mode is complete and will exit
* If there is one next node, ask the user to hit any key to continue
* If there are multiple next nodes, present a menu to the user populated with node conditions to allow the user to choose how to proceed

To demonstrate the stepping mode, UML for how to make a sandwich was loaded into the program. As the figure below demonstrates, the program prints the current step. If there is only one next node, it asks the user to hit any key to continue. If there are multiple next nodes, the user is given a menu with choices for the next node. If there are no next nodes, the user is notified that an end has been reached.

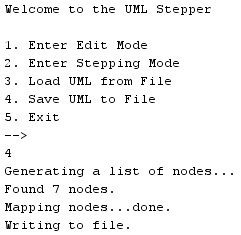


## Save/Load UML Deliverable

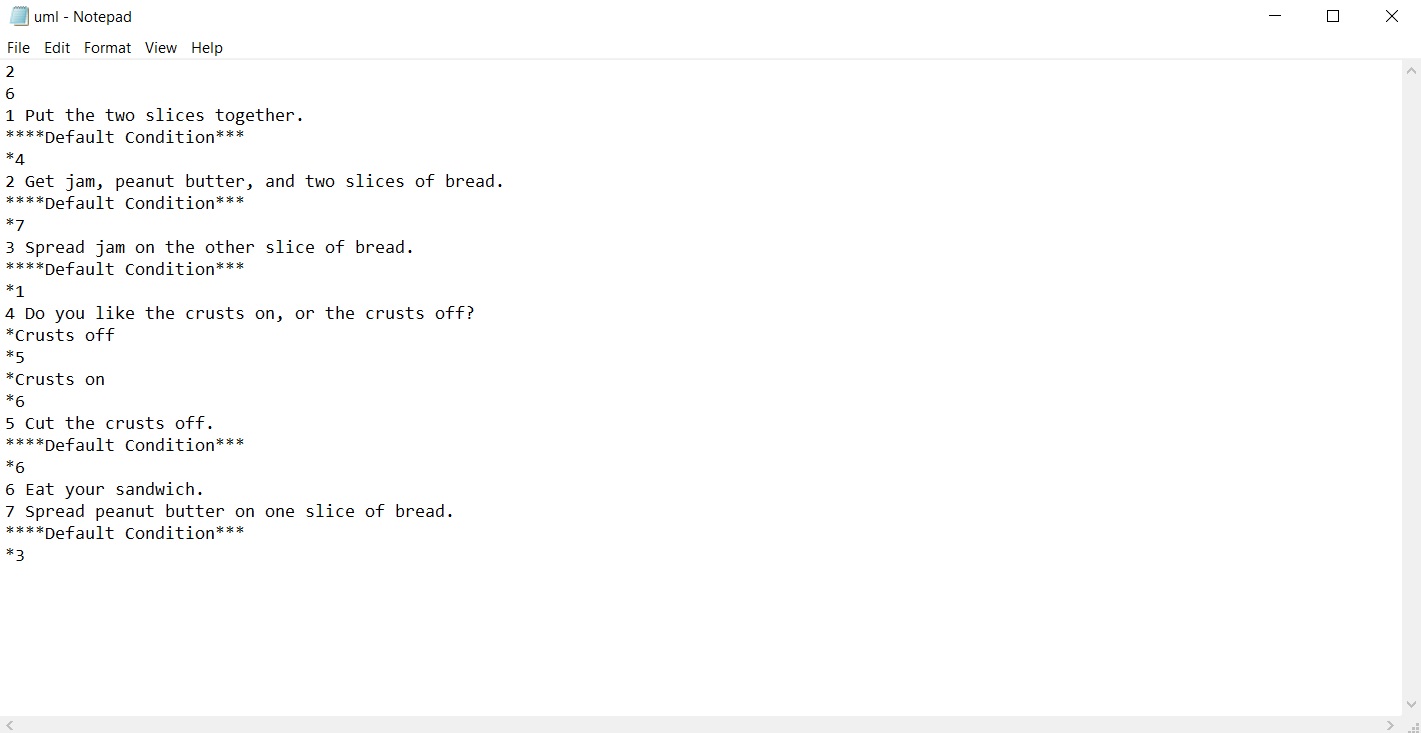
The checklist for the Save/Load UML deliverable is as follows:

* Allow the user to save UML information to a text file
* Allow the user to load UML information from a text file

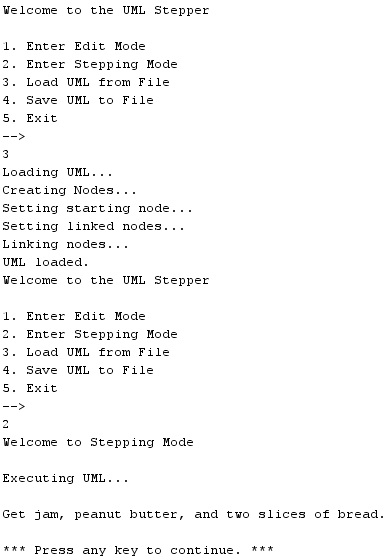
When the user wishes to save, they simply go back to the main menu and choose the “Save UML to File” option this is shown in the figure below.



The program saves the UML in a text file. In the figure below, the sandwich UML was saved. This was the result.



The final component of this program was to have the ability to load UML from a text file. This is demonstrated in the figure below.



# References

Campbell, M. & Hinds, D. (1995). Object-oriented perspective on software system testing in a distributed environment. Hewlett-Packard Journal, 46(6), 75.

Drusinsky, D. (2006). *Modeling and verification using UML statecharts: A working guide to reactive system design, runtime monitoring and execution-based model checking*. Burlington, MA: Newnes.

Levenick, J. (2006). *Simply java:* *An introduction to java programming.* Hingham, Mass: Course PTR.

Rowe, S. (2015). *Project management for small projects, second edition*. Tysons Corner, Virginia: Management Concepts, Inc.