|  |
| --- |
| St Brendan’s sixth form College |
| Critical path Teaching tool |
| AQA Computer Science NEA coursework |

|  |
| --- |
| Candidate Name: |
| Candidate Number: |

Contents

[1 Analysis 3](#_Toc516162092)

[1.1 Introduction 3](#_Toc516162093)

[1.1.1 Problem/Investigation description 3](#_Toc516162094)

[1.1.1.1 Description 3](#_Toc516162095)

[1.1.1.2 Background 3](#_Toc516162096)

[1.1.2 Third party 3](#_Toc516162097)

[1.2 Summary of investigation / research findings and sources 3](#_Toc516162098)

[1.2.1 Interview 3](#_Toc516162099)

[1.2.2 Research 5](#_Toc516162100)

[1.2.3 Similar systems 6](#_Toc516162101)

[1.2.4 Current revision resources 7](#_Toc516162102)

[1.3 IPOS (Input Process Output Storage) 10](#_Toc516162107)

[1.4 Data Dictionary (key variables) 11](#_Toc516162108)

[1.5 Requirements (specification) 11](#_Toc516162109)

[1.5.1 Core Objectives 12](#_Toc516162110)

[1.5.2 Extension Objectives 13](#_Toc516162111)

[1.6 Development planning 13](#_Toc516162112)

[1.6.1 Development methodology 13](#_Toc516162113)

[1.6.2 Critical path analysis 14](#_Toc516162114)

[1.6.3 Problem modelling 14](#_Toc516162115)

[2 Design 15](#_Toc516162116)

[2.1 HCI (Human Computer Interface) 15](#_Toc516162117)

[2.2 Data modelling 15](#_Toc516162118)

[2.3 Data structures 15](#_Toc516162119)

[2.4 Algorithms 15](#_Toc516162120)

[2.5 Program structure 15](#_Toc516162121)

[2.6 File structure and organisation 15](#_Toc516162122)

[2.7 Database design 15](#_Toc516162123)

[2.7.1 ER diagrams 15](#_Toc516162124)

[2.7.2 Queries (SQL design) 15](#_Toc516162125)

[2.8 Hardware selection/design 15](#_Toc516162126)

[3 Technical solution 16](#_Toc516162127)

[3.1 Overview 16](#_Toc516162128)

[3.2 UML diagram(s) 16](#_Toc516162129)

[3.3 Techniques used (referenced to code) / difficult to understand sections 16](#_Toc516162130)

[3.4 Annotated code 16](#_Toc516162131)

[4 Testing 17](#_Toc516162132)

[4.1 Prototype testing 17](#_Toc516162133)

[4.2 Alpha testing 17](#_Toc516162134)

[4.3 Beta testing 17](#_Toc516162135)

[5 Evaluation 18](#_Toc516162136)

[5.1 Objectives 18](#_Toc516162137)

[5.1 Third party feedback and analysis 18](#_Toc516162138)

[5.2 Future improvements 18](#_Toc516162139)

[6 Appendices 19](#_Toc516162140)

# Analysis

## Introduction

## Problem/Investigation description

### Description

This project is about creating a teaching tool which can be used to help A level further maths students understand how the critical path algorithm works to help with their end of year 2 exams

### Background

The critical path algorithm is currently in the Edexcel A level further maths syllabus. It often appears in exams as a multi-part question which could involve any of the following parts: Filling out the early starts and late finishes, calculating the floats for tasks, finding the critical path(s), calculating the earliest time for the task to be completed, coping with delays within the tasks, and coping with scheduling tasks which require special equipment. Edexcel uses the activity on edge convention, as opposed to activity on node.

Solving all of these problems requires the student to understand both the algorithm, and how certain factors can affect the solution.

## Third party

**Louis Theobald:**

Further maths student who wants to build his knowledge about the critical path algorithm to help him achieve his target grade

## 1.2 Summary of investigation / research findings and sources

### 1.2.1 Interview

**Louis Theobald – St Brendan’s Sixth Form College – 05/06/18**

1. **The prototype currently allows you to place a task by clicking a button located at the bottom of the window, could you suggest any other methods to place a task, which you would prefer?**

I like the button, but I would prefer if there was an option to add a task with a right-click menu. The tasks should be able to be dragged and dropped.

1. **What is your opinion on the look and colour scheme of the program?**

The white is clean; it will be good to be able to switch themes to make the background darker, as long as the colours are kept simple. I think the final version should look more professional. The design should remain intuitive.

1. **How would you like the nodes and tasks to look in the final program?**

Try to follow how they look in the revision booklets

1. **Would you as a user prefer to manually name the nodes, or for them to automatically be labelled?**

It would be simpler and quicker if the nodes be named starting from 0 and counting up automatically.

1. **Should the program be adjusted to allow the user to pan the window, and zoom in and out, allowing larger networks to be constructed?**

It would be a good extension but I don’t think it’s necessary.

1. **The final program will allow the user to input values for the early and late start, it will then be able to check these inputs to tell the user if they are correct. How do you think the program should tell the user if they have put a correct value?**

There should be a button which checks all the answers, this option could become selectable once everything has been filled in or there could be a mode where the button can be clicked at any time, if there’s any answers not filled in they shouldn’t be revealed.

1. **Following on from this, should the user be marked on how much they get correct?**

I think it’s a good idea, as long as the marks are reflective of what the questions will give in the exam

1. **Should there be a few example networks included?**

There could be a few examples but I don’t think it’s necessary.

1. **How should dummy tasks be added?**

I think there should be a separate option to add a dummy task, or adding a task and setting the duration to 0 should give it the dummy look.

1. **How would you like the critical path to be distinguished from the other paths?**

The tasks on the critical path should be a different colour to the others; they should only become that colour when the answers are shown or when identified by the user.

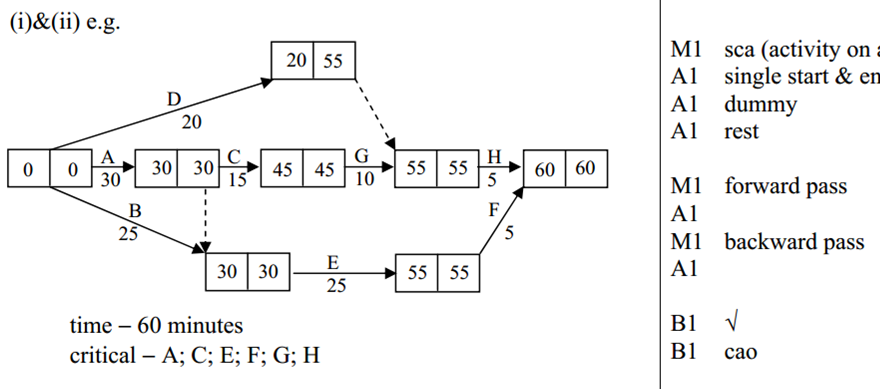
1. **The final project will allow the user to delete tasks, and delete nodes. Also, the window will be automatically scaled to the user’s monitor. Are there any other features not mentioned which you would like to be included in the final project?**

Maybe you could add a mode to create Gantt charts out of the networks. Also, change the duration of a task. And, if you have time make the window re-scalable.

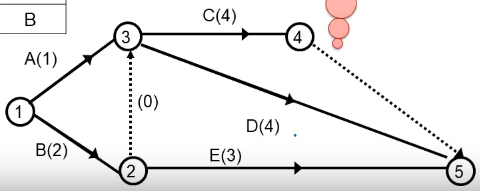
### 1.2.2 Research

The critical path method (CPM) was developed originally by DuPont to optimise the complex task of shutting down chemical plants for maintenance and restarting them after maintenance1

The CPM needs a list of tasks, for each task the computer/manager needs to know which other tasks have to be completed before it can be started, and the duration of the task. The tasks can then be mapped on a diagram as lines which are connected to nodes, a node is a point which one task finishes and other tasks begin, in the current specification for the A level, a critical path diagram would look like this:

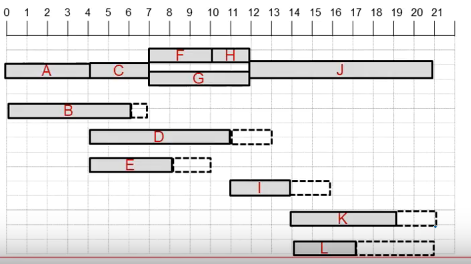
 2

In the diagram shown above, the nodes are represented by the circles, and the tasks are the arrows. The dotted arrows are called “dummy” activities, which are activities with duration of zero; they are used when sets of tasks share some, but not all, prior activities. An example of this is the dummy between the node at the end of B, and the node at the end of A. This is used here since activity E requires both A, and B to be completed before it can be started, whilst activity C only requires A.

3

They are also used when multiple tasks have the same start and end node, in this case one of the tasks is made to lead to another node. This new node will then have a task which has a duration of 0 (a dummy task), linking back to the node where it would normally finish. An example of this behaviour in the diagram is the relationship between task C, task D, and nodes 4 and 5.

This path can be used to create a Gantt chart to show how many workers are needed to complete a project, as well as which jobs are being completed at a certain time. The Gantt chart was originally created by American engineers Henry Gantt and Frederick Taylor, as a graphical way to illustrate project schedules.4 A Gantt chart can be drawn from the information given by CPM to see how many workers are needed for a task, as well as when each task needs to start and when it will finish, making it of paramount importance in business. Below is an example of the type of Gantt chart used for further maths A level.



5

**Sources:**

1: NetMBA.com, "CPM - Critical Path Method," Process Flow Structure, , accessed May 09, 2018, <http://www.netmba.com/operations/project/cpm/>.

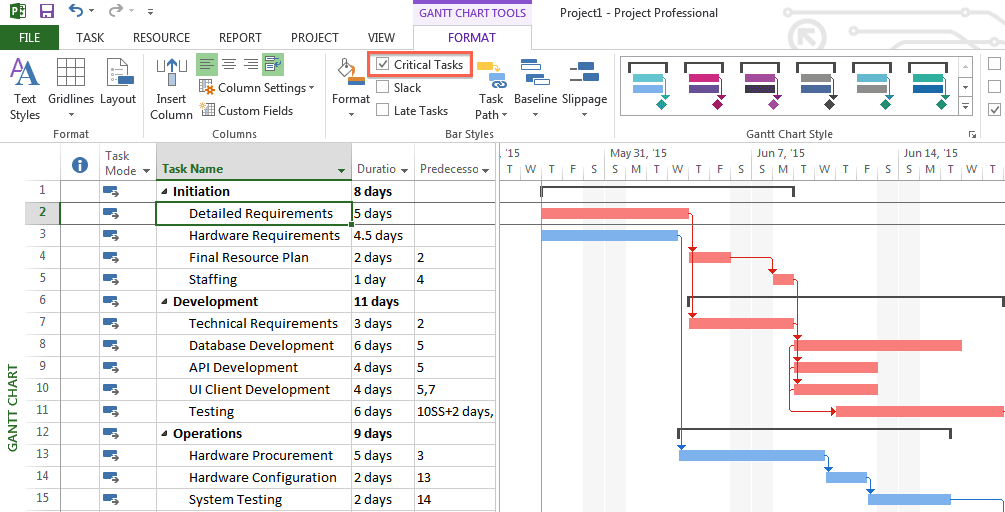
2: <https://www.thestudentroom.co.uk/attachment.php?attachmentid=291520&stc=1&d=1401792310>

3: <https://www.youtube.com/watch?time_continue=1&v=1XCgOwIsrYw>

4: "What Is a Gantt Chart?" Gantt. Accessed May 22, 2018. http://www.gantt.com/.

### 1.2.3 Similar systems

Microsoft Project: image - https://www.smartsheet.com/critical-path-method



Good features:

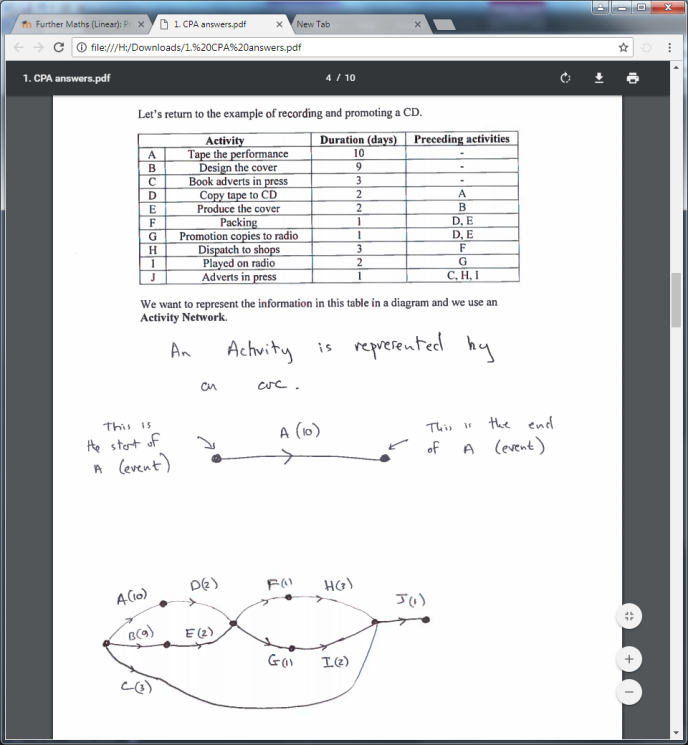
* Used in industry so has a professional interface
* Interchangeable themes
* Option to show critical tasks clearly
* Draws a Gantt chart with clear colours
* Many functions and options which can change what information is displayed
* Simple to enter the predecessors for each task
* Allows manual rescheduling of the activities

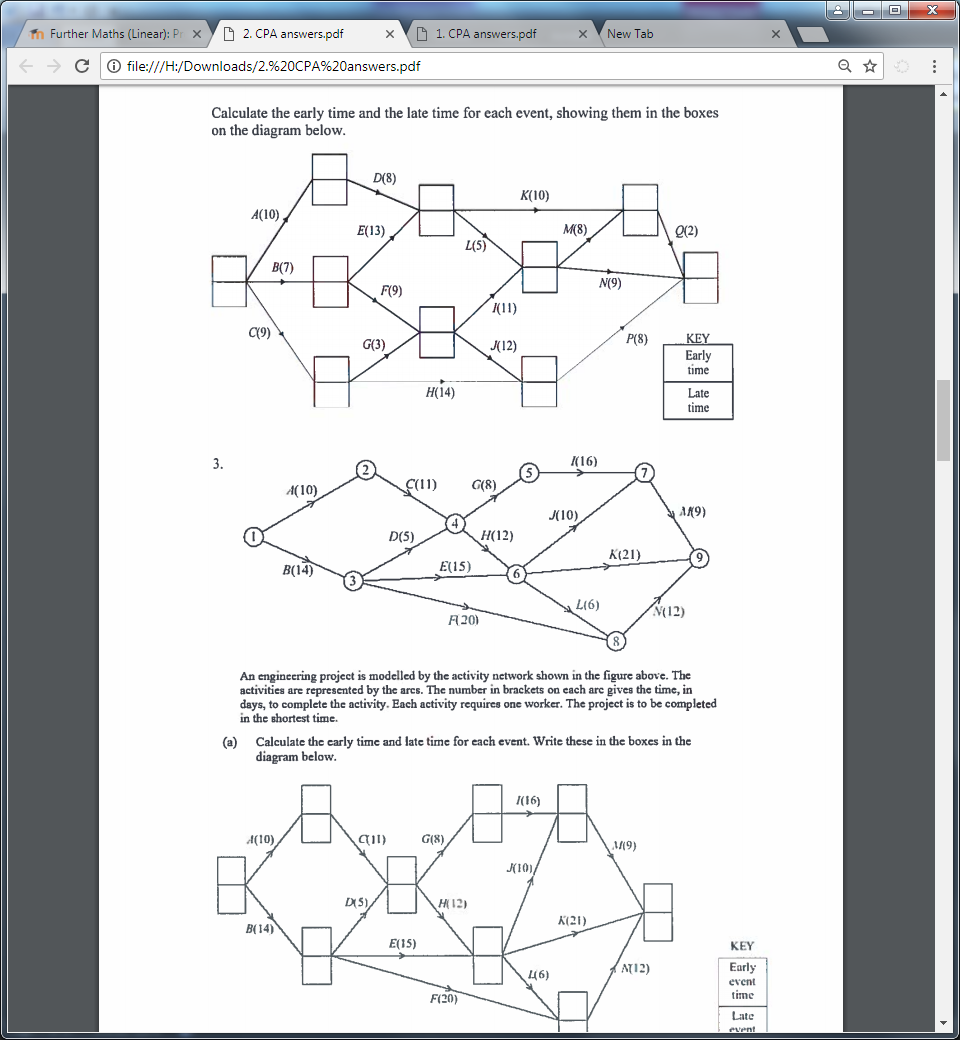
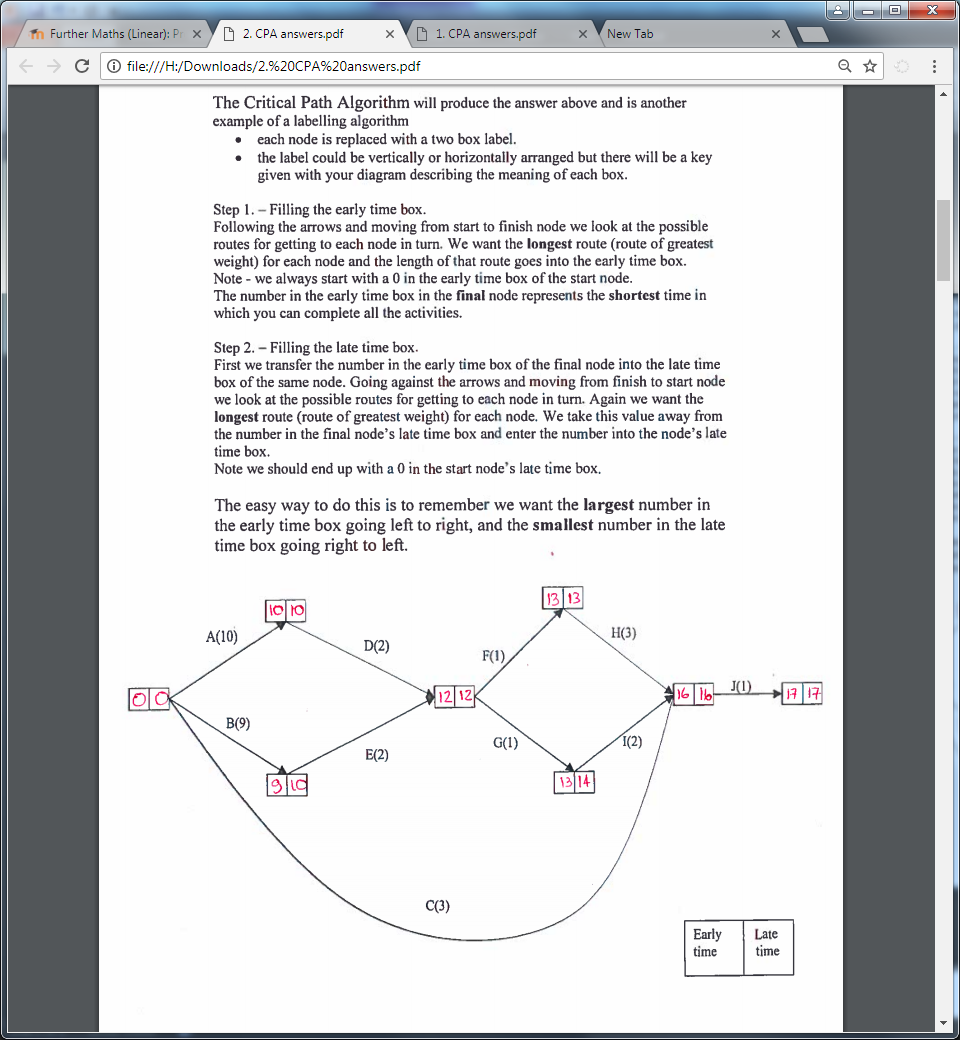
Bad features:

* Could look complicated to people learning the algorithm
* Doesn’t allow the user to fill out any diagram
* Doesn’t show the critical path in a diagram
* Part of Microsoft Office, which is paid software

I could not find any other similar systems, so there is a definite need for more programs like this

### 1.2.4 Current revision resources

****Current college resource to teach the critical path algorithm and precedence tables:



## Good features:

## Clear structure making it easy to follow for a reader

## Explained well with hand written solutions

## Gives the student questions to answer with worked solutions

* Looks exactly how the questions will look in the exam

Bad features and limitations:

* Only a limited amount of questions
* Answers have to manually be checked
* Can’t distinguish between critical path and other paths
* The user prefers to work at a computer instead of on pen and paper

## IPOS (Input Process Output Storage)

|  |  |  |  |
| --- | --- | --- | --- |
| **Inputs** | **Processes** | **Outputs** | **Storage** |
| Task length | Calculating early start times | Visual tasks | Task length |
| Task’s dependencies | Calculating late finish times | Visual nodes | Task’s dependencies |
| User’s early start times | Checking users early start times against the calculated values | Whether each inputted answer is correct | User’s inputs |
| User’s late start times | Checking user late finish times against calculated values | Box to put early and late start in | List of all the tasks |
| Where to place tasks | Building and displaying main path window | Inputted text | Position of tasks |
| Right click to open context menu | Building and displaying context menu | Context menu | Calculated values for late and early start |
| User inputs for tasks and dependencies in table | Calculating float times for activities | Table of tasks and dependencies | Position of table |
|  |  |  | Position of nodes |

## 1.4 Data Dictionary (key variables)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Purpose** | **Data type** | **Example** | **Validation/size** |
| name | Stores the identifier for the task | String | “A” | Alphanumeric, not null |
| Dependencies | Holds a list of which tasks the task is dependent on | Task array | A,B,D | Size greater than equal to 0 |
| earlyStartCalculated | Holds the early event time for the task | Integer | 7 | Size greater than or equal to 0 |
| latestFinishCalculated | Holds the late event time for the task | Integer | 9 | Size greater than or equal to 0 |
| earlyStartInput | Holds the user’s input for the early event time, to be checked against the calculated value | Integer | 7 | Any size |
| latestFinishInput | Holds the user’s input for the late event time, to be checked against the calculated value | Integer | 16 | Any size |
| float | Stores the total float for the task | Integer | 3 | Size greater than or equal to 0 |
| CriticalCond | Stores whether a task is critical or not | Boolean | true | True or false |

## 1.5 Requirements (specification)

(Q5) means the objective is related to question 5 of the interview

### 1.5.1 Core Objectives

CO1: The user must be able to create a task, specifying the length of the task as well as its location (Q1)

CO2: The user must be able to create a node, which will be given a unique identifier; the first node will have an identifier of 1, the second 2, and so on…. The user must be able to specify a nodes location, as long as it doesn’t intersect another node (Q4)

CO3 Placed nodes must be able to be dragged across the component, attached tasks must move with the nodes

CO4: The diagram of the tasks must be in a format similar to the taught format for the A level specification, which wouldn’t alienate the user (Q3)

CO5: The user must be able to input their calculated values for the early and late event times for each node

CO6: The program must have a method of calculating the early and late event times for every node in any given network

CO7: The program must have a method of calculating which tasks are critical in any given network

CO8: The program must have a method of calculating the critical path length of any given network

CO9: The program must have a method of showing the user the correct early starts and the correct late finishes for all connected nodes, making clear whether the user inputted correctly (Q7)

CO10: The program must have a method of allowing the user to identify tasks as critical, and showing whether tasks are identified correctly (Q10)

CO11: The user must be able to edit the length of tasks

CO12: When creating tasks, a task must only be allowed if it starts and ends at a node (Q3)

CO13: The user must be able to create a dummy task, which will be drawn as a dashed line, making it distinguishable from regular tasks (Q4)

CO14: The program must have an option to verify whether the diagram which the user has drawn is valid, this would verify that all the tasks are linked to nodes, that there are no tasks starting and finishing at the same nodes as another task, and that the path contains no cycles. (Q7)

CO15: The layout of the program must not be over packed; it must be intuitive and easy to learn the functions, measured by the opinion of the user (Q2)

CO16: The program must look professional in the opinion of the user (Q2)

CO17: The program must scale to the size of the user’s monitor automatically

CO18: The user must be able to delete nodes and tasks, deleting a node must delete directly connected tasks

### 1.5.2 Extension Objectives

EO1.1: The program could be altered to add a new option which allows the user to switch to a Gantt chart view, which shows the added activities in the form of a scheduling chart (Q12)

EO1.2: The added Gantt chart mode will allow the user to move activities within their floats

EO1.3: The tasks on the Gantt chart should be moveable by the user onto different rows, allowing the user to practice rescheduling activities

EO2: For the program to give the user marks out of a total when they check their answers, giving the user more incentive to learn and improve. The amount of marks awarded should be accurate in respect to how many marks are awarded and what merits each mark (4 marks for correct event times, 2 marks for stating critical activities) (Q8)

EO3: For the user to have the option to change the theme of the program, this would adjust background colour as well as the colour of the tasks (Q2)

EO4: The window of the program may be altered to become re-scalable; re-scaling must keep the UI intact and all options viewable (Q11)

EO5: The program could have a few pre-made paths to show the user examples of what they could make (Q9)

EO6: The program could be adjusted to pan the task window, and zoom in and out of it (Q6)

EO7: The program could be lenient and allow multiple networks to be created and marked at one instance

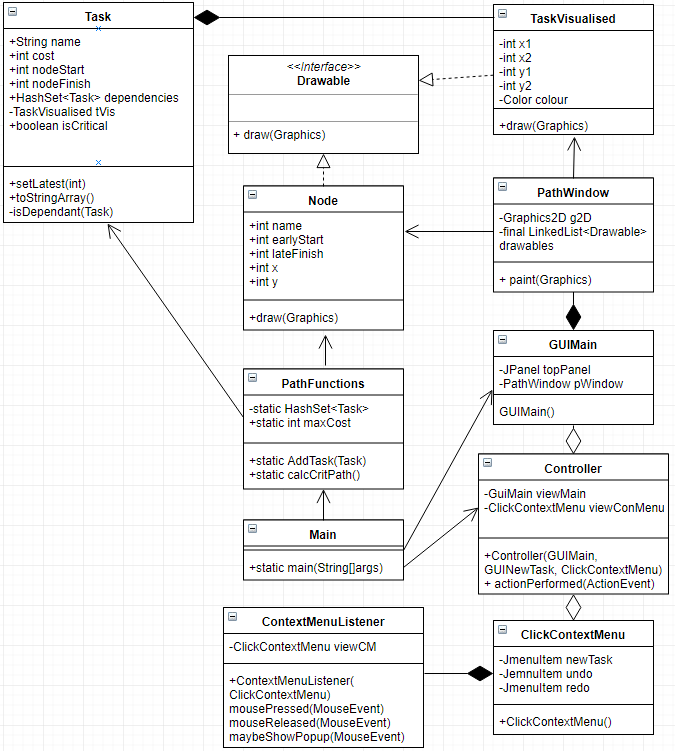
## 1.6 Development planning

### 1.6.1 Development methodology

I will apply an agile approach to the development of this project

### 1.6.2 Critical path analysis

### 1.6.3 Problem modelling

OOP diagram for how the solution could be modelled:

# 2 Design

### 2.2.1 Class definitions

### 

### 

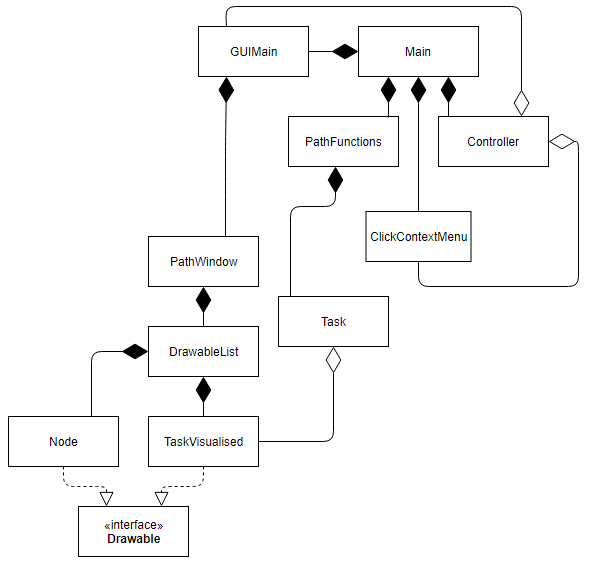
### 

### 

### 

### 

### 2.2.2 Object diagram



## 2.3 Data structures

The program has one non-default interface, the ‘Drawable’ interface. The purpose of this interface is to allow the objects which implement it, (the visualisedTasks, and nodes), to be stored in one list to be drawn in the graphical window. Each drawable object has a ‘draw’ procedure, which has a graphics object as a parameter.

Following on from this, DrawableList is a linked list of drawable objects. The purpose of this list is to join the visualisedTasks and nodes into one list of every object which needs to be painted into the main window. I have chosen to use a linked list implemented using objects instead of other data structures since they are dynamic, meaning the list does not have a fixed length. It is important for this data structure to be dynamic because items needed to added and removed so there cannot be a pre-determined length.

In addition, the list needs to draw items one at a time in order of tasks first, and then nodes. This is because the nodes need to be drawn over the tasks so the tasks can go to the centre of the nodes without being visible. Having a method, which goes through the list from start to finish, drawing each item in order, will draw the later items, the nodes, on top of the first items, the tasks. This allows everything to overlap correctly. The DrawableList uses a method called ‘drawAll’, to do this.

A linked list is ideal when objects need to be sorted by priority, since the next item in the list is referenced by the previous. Therefore, if a task is added, just by changing which object references the first node, it can be easily placed into the list before the first node. Each item within the list has a ‘priority’ constant, which is equal to 1 for a taskVisualised, or 2 for a Node. By using the insert procedure, an item can be added to the list. The procedure iterates through the list, checking if each object’s priority is equal to or greater than the new object’s priority. If it is then the object is placed in that position, otherwise the procedure keeps iterating through until this condition is met.

I also have created another linked list, the NodeList, which is a list of every created node. This list’s purpose is to check if the user is trying to drag a node into another node, and to run through every node and check the inputted answers when the user wants to check their answers. Finally, it can see if a click location is within any node, and return which node contains the click.

I chose to make a linked list of nodes since there should not be a maximum amount of nodes, so I needed a dynamic data structure. The NodeList is implemented the same way as the DrawableList, except it does not sort by priority.

I have used a few HashSets in my solution; I used them in areas where I need to hold multiple tasks. For example, when running through the algorithms to find early and late event times, the tasks need to be stored in either a completed set, or a remaining set. I chose a set because each task needs to be unique, and a set will not hold duplicate objects. I chose to use a HashSet instead of other types of set because there doesn’t need to be an order to how the tasks are held.

## 2.4 Algorithms

**1: Drawing a task:**

Draw method for task. ‘aboveMiddleOfTask’ means a location where the perpendicular distance from the middle of the task to a point is equal to a set value. To find this point, I use an algorithm similar to algorithm 12, instead of returning a line it returns a co-ordinate to the end of a line of given length, perpendicular to the midpoint. The colour will be blue if a task has been identified as critical, or red if any of the following conditions are met: If a task has been identified as critical and is not critical. If a task has not been identified as critical, but is critical. Alternatively, a brighter red if the task has been selected.

SetColour

DrawLine(start to finish)

DrawText(Duration at location: aboveMiddleOfTask)

**2: Drawing a node:**

Draw method for node

DrawCircle(centre, radius)

**3: Drawing all the drawable objects:**

Loops through each object in the list, calling its draw method

Current Drawable 🡨 first

WHILE current not null

Draw current

Current 🡨 next Drawable

ENDWHILE

**4: Calculate the critical costs and all other task information from a set of tasks:**

This is the main path-finding algorithm in the program, and requires explanation:

The algorithm starts by creating the ‘completed’ set, which holds the tasks whose critical costs have been calculated. In addition to the ‘remaining’ set, this holds the rest of the tasks. Initially, ‘remaining’ holds all the tasks. The first ‘for’ and ‘while’ loops iterate through every task still in ‘remaining’, until ‘remaining’ is empty. On each iteration, the code within the loop tries to calculate the criticalCost for the current task. To calculate the criticalCost, every task which the current task is dependent on needs to have a calculated criticalCost. To verify this, an ‘if’ statement checks if all the dependencies are in the completed set. If they are not, the loop re-iterates with the next task. However, if the statement returns true, the ‘critical’ variable is set to the largest criticalCost of all the dependencies. Then the criticalCost of the current task is calculated, and the task can now be marked as completed. So it is added to completed, and removed from remaining. Progress is marked as true if this happens. If no progress is made over the whole of the ‘for’ loop, then the path must contain a cycle, which means there is no path from start to finish.

All the rest of the information about the tasks can now be calculated, because critical costs are known; and once everything has been calculated, the visual element of the tasks can be updated

completed 🡨 new Task set

remaining 🡨 new Task set

remaining 🡨 Set of all Tasks

WHILE remaining !Empty

progress 🡨 false

FOR i 🡨first remaining to last remaining

currentTask 🡨 next i

IF completed containsAll( t dependencies)

INTEGER critical 🡨 0

FOR i 🡨 0 to amount of dependancies

IF t criticalCost > critical

critical = t criticalCost

ENDIF

ENDFOR

t criticalCost = critical + taskCost

completed add(t)

remaining remove(t)

progress 🡨 true

ENDIF

ENDFOR

if !progress

return null

ENDWHILE

maxCost (task Set) (see algorithm 5)

initialNodes 🡨 newTaskSet

initialNodes 🡨 findInitials(tasks) (See algorithm 7)

CalculateEarly(initalNodes) (see algorithm 8)

return tasks[]

**5: Calculate length of critical path (max Cost algorithm):**

This method iterates through the complete list of tasks and finds which task has the greatest criticalCost, since the greatest critical cost is equal to the critical path length. The path length is now known. This information is also necessary to find the latest start and finish of each task.

max 🡨 -1

FOR i 🡨 0 to amount of tasks

TASK t 🡨 taskList[i]

IF t criticalCost > max

max 🡨 t criticalCost

ENDIF

ENDFOR

maxCost 🡨 max

FOR i 🡨 0 to amount of tasks

TASK t 🡨 taskList[i]

t setLatest(maxcost) (See algorithm 6)

ENDFOR

**6: Calculate latest start and latest finish of a task (SetLatest algorithm):**

Quick calculation using the max cost to find latest start time, and then a calculation using latest start to find latest finish time.

latestStart 🡨 maxCost - criticalCost

latestFinish 🡨 latestStart + cost

**7: Find which task(s) are at the beginning of the path (Find initials algorithm):**

A task, which is dependent on nothing, is an initial task

remaining🡨new Task set

remaining 🡨 Set of tasks

FOR i 🡨 0 to amount of tasks

TASK t = tasks[i]

FOR iD = 0 to amount of t dependancies

remaining remove(t dependencies[iD])

**8: Calculate the early starts for initial tasks:**

Simply runs a loop to set the early start and finish for the initial nodes, then runs an algorithm to work through the rest of the tasks calculating their early starts and finishes

FOR i 🡨 0 to amount of initial tasks

TASK t = initials[i]

t earlyStart 🡨 0

t earlyFinish 🡨 t cost

completionTime 🡨 t cost

setEarly(t)

**9: Calculate the early start times for remaining tasks:**

A recursive algorithm, which works through each task, using the initial task as a starting point

completion time = t earlyFinish

FOR j 🡨 0 to amount of tasks dependant on t

TASK current = tasks[j]

IF completionTime >= current earlyStart

current earlyStart 🡨 completionTime

current earlyFinish 🡨 completionTime + current cost

ENDIF

SetEarly(t) (Recursive call)

**10: Check whether a location lies within any of the nodes:**

A function within the NodeList class, which checks every node to see if a click location intersects with a node, the excluded node is not checked

current 🡨 First Node

WHILE (current != null)

IF (current != exluded)

magSquared🡨 |click to centre of node|2

IF magSquared <= radius2

Return current

ENDIF

ENDIF

ENDWHILE

Return null

**11: Make a line segment, which intersects a given task at a given angle:**

This algorithm uses trigonometry to create a line segment, which intersects a task at a point, which is a predefined fraction of the way across the segment. This fraction is called ‘mult’. The new segment, intersects the task at acute angle ‘angle’. The algorithm works in the following way:

* Find the point on the task where the new line will begin
* Get the vector joining the start of the node to the task
* Convert the joining vector into polar co-ordinates
* Add the angle of the polar line to the inputted angle of intersection
* Find the x coordinate for the end of the new line
* Find the y coordinate for the end of the new line
* Return the line joining the start of the line to the end coordinates

point 🡨 ((end – start) \* mult) + start

startToPoint 🡨point – start

theta 🡨polar angle(startToPoint)

rho 🡨theta + angle

x2 🡨xPoint - (length \* cos(rho))

y2 🡨yPoint – (length \* sin(rho))

Line 🡨 (point to (x2,y2))

**12: Find the perpendicular distance from a task to a given vector:**

If you had a line segment and a point ‘v’, you can use vector mathematics to find the shortest distance between the point and the line. The way this works, is by finding another line, which starts at the point and intersects the original segment perpendicularly. Note, the calculated distance is the squared distance, to avoid the slow Math.sqrt() function. This function is used to see if a user has selected the task. The process is as follows:

* Find the vector from the start of the task to the point
* Get the vector from the start of the segment to the end of the segment
* Verify that the line has a length greater than 0
* Find the dot product of the line segment and the vector from the start of the line to the point
* Divide this dot product by the line’s magnitude squared, and store as a decimal
* Multiply the length of the line by the decimal, and add this to the start of the line, to get the vector from the origin to the closest point on the line
* Subtract this vector from the vector from the origin to the point, giving the shortest vector from the point to line
* Return the squared magnitude of this vector

startToPoint 🡨 v – start

line 🡨 end – start

p 🡨 20

IF |line|2 != 0

p 🡨 (startToPoint · line) / |line|2)

ENDIF

d 🡨 v – (start + (line \* p))

return |d|2

## 2.5 Program structure

### 2.7.1 ER diagrams

text

# 3 Technical solution

## 3.1 Overview

text

## 3.2 UML diagram(s)

See Design

## 3.3 Techniques used (referenced to code) / difficult to understand sections

* Recursion: PathFunctions – setEarly method
* Custom Data Structures – NodeList, DrawableList
* Custom Interface: Drawable
* Casting custom objects: PathComponent – dragSelected and ctrlClick (Also in DrawableList and GUIMain)
* Advanced mathematical functions: Vector class
* Application of critical path algorithm: PathFunctions
* Custom error Throwing: PathFunctions – criticalPath method
* Try Catch Statements: Main - main

## 3.4 Annotated code

Main

**import** javax.swing.\*;  
  
*/\*\*  
 \* Main class, simply creates the objects needed for the program to run  
 \*/***class** Main {  
 */\*\*  
 \* Main method where Look and feel of GUI, the GUI, the Controller,  
 \* and the context menu are created  
 \** ***@param args*** *- Array of arguments  
 \*/* **public static void** main (String[]args){  
 **try** {  
 UIManager.*setLookAndFeel*(UIManager.*getCrossPlatformLookAndFeelClassName*());  
 UIManager.*setLookAndFeel*(**"com.sun.java.swing.plaf.nimbus.NimbusLookAndFeel"**);  
 } **catch**(Exception e){  
 System.***out***.print(**"No Layout Manager"**);  
 }  
 GUIMain mainGui = **new** GUIMain();  
 ClickContextMenu cCMenu = **new** ClickContextMenu();  
 **new** Controller(mainGui, cCMenu);  
 }  
}

GUIMain

**import** javax.swing.\*;  
**import** java.awt.\*;  
*/\*\*  
 \* GUI object, which extends JFrame  
 \* Holds the Component for the Nodes and Tasks, and all the options  
 \* Class is initialised by the Main class  
 \*/***class** GUIMain **extends** JFrame {  
 **private final** JButton **CHECK\_BUTTON** = **new** JButton(**"Check Answers"**);  
 **private final** JButton **CHECK\_VALID\_BUTTON** = **new** JButton(**"Check Validity Of Network"**);  
 **private final** JButton **CLEAR\_BUTTON** = **new** JButton(**"Clear all"**);  
 **private final** PathComponent **P\_COMPONENT** = **new** PathComponent();  
 **private final** JLabel **CRIT\_PATH\_LENGTH**;  
 **private final** JLabel **MARK\_LABEL**;  
  
 */\*\*  
 \* Constructor for class  
 \* Configures the frame  
 \*/* GUIMain(){  
 **P\_COMPONENT**.setPreferredSize(**new** Dimension(700, 500));  
 setDefaultCloseOperation(***EXIT\_ON\_CLOSE***);  
 JPanel bottomPanel = **new** JPanel();  
 JPanel topPanel = **new** JPanel();  
 **CRIT\_PATH\_LENGTH** = **new** JLabel();  
 **MARK\_LABEL** = **new** JLabel();  
 bottomPanel.add(**CLEAR\_BUTTON**);  
 bottomPanel.add(**CHECK\_VALID\_BUTTON**);  
 bottomPanel.add(**CHECK\_BUTTON**);  
 **CRIT\_PATH\_LENGTH**.setAlignmentX(Component.***RIGHT\_ALIGNMENT***);  
 bottomPanel.add(**CRIT\_PATH\_LENGTH**);  
 topPanel.add(**MARK\_LABEL**);  
 **this**.setTitle(**"Critical Path Algorithm"**);  
 **this**.getContentPane().add(topPanel, BorderLayout.***NORTH***);  
 **this**.getContentPane().add(**P\_COMPONENT**, BorderLayout.***CENTER***);  
 **this**.getContentPane().add(bottomPanel, BorderLayout.***SOUTH***);  
 **this**.pack();  
 **this**.setVisible(**true**);  
 **this**.setFocusable(**true**);  
 **this**.requestFocus();  
 }  
  
 */\*\*  
 \* Function to initiate the the drawing of a task  
 \* Checks if location is valid  
 \* Starts drawing the line  
 \** ***@param v*** *- The location of the start of the line  
 \** ***@return*** *- The start node  
 \*/* Node startDrawing(Vector v){  
 Node startNode = **P\_COMPONENT**.validTaskStart(v);  
 **if** (!(startNode == **null**)){  
 **this**.**P\_COMPONENT**.drawLine(startNode.getCentre());  
 **return** startNode;  
 }  
 System.***out***.println(**"Invalid start"**);  
 **return null**;  
 }  
  
 */\*\*  
 \* Procedure to process a double click  
 \* First checks if a Drawable is selected  
 \* If a TaskVisualised is selected, then opens a dialogue box to set duration  
 \* Else if a Node is selected, then the selected zone of the node is checked,  
 \* allowing the correct dialogue box to open to change the correct event time  
 \* When a dialogue box receives an input, the input is checked and set to the corresponding variable if valid  
 \* If no node is selected then no dialogue box is opened  
 \*/* **void** doubleClicked (){  
 String input = **"null"**;  
 Drawable selectedDrawable = **P\_COMPONENT**.getSelected();  
 **if** (selectedDrawable != **null**) {  
 **try** {  
 **if** (selectedDrawable.getObjectPriority() == 1) {  
 TaskVisualised tVis = (TaskVisualised) selectedDrawable; *//Safe to cast because selectedDrawable is proven to be a taskVisualised* input = JOptionPane.*showInputDialog*(**"New duration for task: "**);  
 **if** ((input != **null**)&&(Integer.*parseInt*(input)>= 0)) {  
 tVis.setDuration(Integer.*parseInt*(input));  
 }  
 } **else if** (selectedDrawable.getObjectPriority() == 2) {  
 Node n = (Node) selectedDrawable;  
 **if** (n.getSelectedArea() == 2) {  
 input = JOptionPane.*showInputDialog*(**"Enter early event time: "**);  
 **if** ((input != **null**)&&(Integer.*parseInt*(input)>= 0)) {  
 n.setEarlyStartInput(Integer.*parseInt*(input));  
 }  
 } **else if** (n.getSelectedArea() == 3) {  
 input = JOptionPane.*showInputDialog*(**"Enter late event time: "**);  
 **if** ((input != **null**)&&(Integer.*parseInt*(input)>= 0)) {  
 n.setLatestStartInput(Integer.*parseInt*(input));  
 }  
  
 }  
  
 }  
 **if** (Integer.*parseInt*(input)>= 0) { *//Output on console whether input is valid or not (Testing)* System.***out***.println(input + **" is valid"**);  
 }  
 **else** {  
 System.***out***.println(input + **" is invalid (Inputs cannot be smaller than 0)"**);  
 }  
 }  
 **catch** (RuntimeException e){  
 **try** {  
 **if** (!input.equals(**"null"**)) {  
 System.***out***.println(input + **" is an invalid input"**);  
 }  
 }  
 **catch** (NullPointerException np){  
 *//Empty invalid, no further action needed* }  
 }  
 }  
 **P\_COMPONENT**.repaint();  
 }  
  
 */\*\*  
 \* Function to get the Paint Component  
 \** ***@return*** *- PaintComponent  
 \*/* PathComponent getPComponent() {  
 **return P\_COMPONENT**;  
 }  
  
 */\*\*  
 \* Function to get the clear button  
 \** ***@return*** *- Clear button  
 \*/* JButton getCLEAR\_BUTTON(){  
 **return CLEAR\_BUTTON**;  
 }  
 */\*\*  
 \* Function to get the check answer button  
 \** ***@return*** *- Check answer button  
 \*/* JButton getCHECK\_BUTTON() {  
 **return CHECK\_BUTTON**;  
 }  
  
 */\*\*  
 \* Function to get the check validity of network button  
 \** ***@return*** *- Check valid button  
 \*/* JButton getCHECK\_VALID\_BUTTON() {  
 **return CHECK\_VALID\_BUTTON**;  
 }  
 */\*\*  
 \* Procedure to set the length of the critical path, which is displayed on the bottom panel  
 \** ***@param critPathLength*** *- The new length to be displayed  
 \*/* **void** setCRIT\_PATH\_LENGTH(String critPathLength) {  
 **this**.**CRIT\_PATH\_LENGTH**.setText(critPathLength);  
 }  
  
 */\*\*  
 \* Sets the GUI to display the total amount of marks achieved  
 \** ***@param marks*** *- Array of marks for respective sections  
 \*/* **void** setMarkAmount(**int**[] marks){  
 **MARK\_LABEL**.setText(Integer.*toString*(marks[0]) + **" / 4 For nodes, "** + Integer.*toString*(marks[1]) + **" /2 For Tasks, "** + Integer.*toString*(marks[0] + marks[1]) + **" /6 Total"**);  
 }  
  
 */\*\*  
 \* Procedure to clear te text on the mark label  
 \*/* **void** clearMarkLabel(){  
 **MARK\_LABEL**.setText(**""**);  
 }  
}

PathComponent

**import** javax.swing.\*;  
**import** java.awt.\*;  
  
*/\*\*  
 \* Class for the PathComponent object  
 \* This object is a custom child class of JComponent  
 \* It is considered as the main component of GUIMain  
 \*/***class** PathComponent **extends** JComponent {  
 **private boolean drawing**;  
 **private int x1**, **x2**, **y1**, **y2**;  
 **private int currentIndex** = 0;  
 **private final** DrawableList **D\_LIST**;  
 **private final** NodeList **N\_LIST**;  
 **private** Node **currentStartNode**;  
 **private** Drawable **selectedDrawable**;  
  
 */\*\*  
 \* Constructor for object  
 \* Initialises the custom linked lists  
 \* Sets up custom key bindings for holding control and pressing delete  
 \*/* PathComponent() {  
 setFocusable(**true**);  
 **this**.**drawing** = **false**;  
 **this**.**D\_LIST** = **new** DrawableList();  
 **this**.**N\_LIST** = **new** NodeList();  
 }  
  
 */\*\*  
 \* Override of the JComponent's paintComponent method  
 \* Simply draws the object by calling the drawObjects procedure  
 \** ***@param g*** *- Graphics object  
 \*/* @Override  
 **public void** paintComponent(Graphics g) {  
 **super**.paintComponent(g);  
 drawObjects(g);  
 }  
  
 */\*\*  
 \* Procedure to draw every Drawable onto the component  
 \* Uses the DrawableList's 'drawAll' procedure to draw the objects  
 \* If a new task is being drawn, draw the task from it's start node to the mouse location  
 \* Draws the arrow onto the task to show its direction  
 \* Draws the selection box if multiple things are being selected  
 \** ***@param g*** *- The Graphics object passed in  
 \*/* **private void** drawObjects(Graphics g) {  
 **D\_LIST**.drawAll(g);  
 **if** (**drawing**){  
 g.setColor(**new** Color(0,0,0));  
 g.drawLine(**x1**, **y1**, **x2**, **y2**);  
 Graphics2D g2d = (Graphics2D) g;  
 g2d.draw(**new** Vector(**x1**, **y1**).getIntersectingLine(**new** Vector(**x2**, **y2**), 0.7, 20, 30)); *//Draws arrow* g2d.draw(**new** Vector(**x1**, **y1**).getIntersectingLine(**new** Vector(**x2**, **y2**), 0.7, 20, -30));  
 }  
 }  
  
 */\*\*  
 \* Function to check whether a potential task start location is valid  
 \* If a click to make a task lies within a node it is considered valid  
 \* The NodeList returns which node contains the vector of the click  
 \* If a node is returned then the task's start is set to the found node and this node is returned  
 \* Otherwise returns null, as location is not a valid task task  
 \** ***@param v*** *- Click Vector  
 \** ***@return*** *- Which node contains the vector  
 \*/* Node validTaskStart(Vector v){  
 Node n = **N\_LIST**.whichNodeContains(v);  
 **if** (!(n==**null**)) {  
 System.***out***.println(**"Valid start"**);  
 **this**.**currentStartNode** = n;  
 **return** n;  
 }  
 **return null**;  
 }  
  
 */\*\*  
 \* Function to change the selected Drawable to a new Drawable  
 \* Un-selects the currently selected drawable  
 \* Selects whatever drawable is clicked on  
 \** ***@param v*** *- Location to be tested  
 \*/* **void** updateSelected(Vector v){  
 **if** (**selectedDrawable**!= **null**) {  
 **selectedDrawable**.setSelected(**false**);  
 **selectedDrawable** = **null**;  
 }  
 **if** (**D\_LIST**.getFirst() != **null**) {  
 **selectedDrawable** = **D\_LIST**.whichDrawableContains(v);  
 }  
 **if** (**selectedDrawable** != **null**){  
 **selectedDrawable**.setSelected(**true**);  
 }  
 repaint();  
 }  
 */\*\*  
 \* Procedure to begin the drawing of a line  
 \* Sets start co-ordinates of the line which is being drawn to the start of the new line  
 \* Sets the end co-ordinates to the start co-ordinates, to give a line of zero length  
 \** ***@param v*** *- Start of new line  
 \*/* **void** drawLine(Vector v){  
 **this**.**drawing** = **true**;  
 **x1** = v.getIntX(); **x2** = v.getIntX();  
 **y1** = v.getIntY(); **y2** = v.getIntY();  
 }  
  
 */\*\*  
 \* Procedure to drag the selected node across the component  
 \* If a line is being drawn, changes its end co0ordinates to the vector location it is being dragged to  
 \* Else, if the selected Drawable is a node, the new position is checked to ensure a node isn't being dragged into another node,  
 \* or through the edge of the component  
 \* If it is not, then the selected nodes location is set to the new location  
 \* The component is then repainted  
 \** ***@param v*** *- The location to drag the drawable to  
 \*/* **void** dragSelected(Vector v){  
 Dimension componentSize = getSize();  
 **if** (**drawing**) {  
 **x2** = v.getIntX();  
 **y2** = v.getIntY();  
 }  
 **else**{  
 **if** (v.getIntX() >= componentSize.getWidth() - 31){  
 v.setX(componentSize.getWidth() - 31);  
 }  
 **if** (v.getIntY() >= componentSize.getHeight() - 21){  
 v.setY(componentSize.getHeight() - 21);  
 }  
  
 **if** (v.getIntX() <= 30){  
 v.setX(30);  
 }  
 **if** (v.getIntY() <= 50){  
 v.setY(50);  
 }  
 **if** (!(**selectedDrawable** == **null**)) {  
 **if** (**selectedDrawable**.getObjectPriority() == 2) {  
 Node selected = (Node) **selectedDrawable**; *//Safe to cast because selectedDrawable is proven to be a node* Node collidingNode = **N\_LIST**.circleIntersectCheck(v, selected.getINDEX());  
 **if** (collidingNode == **null**) {  
 selected.setCentre(v);  
 }  
 **else** {  
 Vector potentialCentre = selected.moveToNode(v, collidingNode); *//Tries to make new centre which doesn't intersect node* **if** (**N\_LIST**.circleIntersectCheck(potentialCentre, selected.getINDEX()) == **null**) {  
 selected.setCentre(potentialCentre);  
 }  
 }  
 }  
 }  
 }  
 repaint();  
 }  
  
 */\*\*  
 \* Function to finish the drawing of a task  
 \* Sets the finish node to the node which the mouse is intersecting  
 \* If the finish node is not valid, tries to make a new Node for the task to finish at  
 \* Else, creates a TaskVisualised to represent the Task  
 \* Inserts the task into the DrawableList  
 \* Creates a new Task using this task visualised  
 \* Adds the task into the HashSets for the nodes it starts and ends at  
 \* Sets the TaskVisualised to selected  
 \* Repaints the component and returns the created task  
 \** ***@param v*** *- The vector for the end of the task  
 \** ***@return*** *- The new task  
 \*/* Task finishDrawing(Vector v){  
 Node nFinish = **N\_LIST**.whichNodeContains(v);  
 **drawing** = **false**;  
 **if** (nFinish == **null**) {  
 **if** (**N\_LIST**.circleIntersectCheck(v, -1) == **null**) { *//Exclude no nodes* nFinish = **new** Node(**currentIndex**, v);  
 **D\_LIST**.insert(nFinish);  
 **N\_LIST**.insert(nFinish);  
 **currentIndex**++;  
 nFinish.setSelected(**false**);  
 }  
 **else** {  
 System.***out***.println(**"Invalid end"**);  
 repaint();  
 **return null**;  
 }  
 }  
 **if** (**currentStartNode** == nFinish){  
 System.***out***.println(**"Task cannot start and end at the same node"**);  
 repaint();  
 **return null**;  
 }  
 TaskVisualised tVis = **new** TaskVisualised(**currentStartNode**, nFinish);  
 **if** (**D\_LIST**.containsTask(tVis)) {  
 System.***out***.println(**"Task already found between these nodes"**);  
 repaint();  
 **return null**;  
 }  
 System.***out***.println(**"Valid end"**);  
 **D\_LIST**.insert(tVis);  
 System.***out***.println(**"Inserted"**);  
 Task t = **new** Task(tVis);  
 tVis.getSTART\_NODE().addToSucceeding(t);  
 tVis.getEND\_NODE().addToPreceding(t);  
 **if** (**selectedDrawable**!= **null**) {  
 **selectedDrawable**.setSelected(**false**);  
 **selectedDrawable** = **null**;  
 }  
 **selectedDrawable** = tVis;  
 repaint();  
 **return** t;  
 }  
  
 */\*\*  
 \* Procedure to process a click whilst the program is in 'node' mode  
 \*  
 \* Creates a node if nothing is selected  
 \** ***@param clickPos*** *- The vector location of the click  
 \*/* **void** nodeModeClick(Vector clickPos){  
 Dimension componentSize = getSize();  
 **if** (clickPos.getIntX() >= componentSize.getWidth() - 31){  
 clickPos.setX(componentSize.getWidth() - 31);  
 }  
 **if** (clickPos.getIntY() >= componentSize.getHeight() - 21){  
 clickPos.setY(componentSize.getHeight() - 21);  
 }  
  
 **if** (clickPos.getIntX() <= 30){  
 clickPos.setX(30);  
 }  
 **if** (clickPos.getIntY() <= 50){  
 clickPos.setY(50);  
 }  
 updateSelected(clickPos);  
  
 **if** (**selectedDrawable** == **null**) {  
 **if** (**N\_LIST**.circleIntersectCheck(clickPos, -1) == **null**) { *//Don't exclude any Nodes from search* Node n = **new** Node(**currentIndex**, clickPos);  
 **D\_LIST**.insert(n);  
 **N\_LIST**.insert(n);  
 **currentIndex**++;  
 **selectedDrawable** = n;  
 repaint();  
 }  
 **else**{  
 System.***out***.println(**"Potential node location intersects node"**);  
 }  
 }  
 }  
  
 */\*\*  
 \* Function to get the selected Drawable  
 \** ***@return*** *- Selected Drawable  
 \*/* Drawable getSelected(){  
 **return selectedDrawable**;  
 }  
  
 */\*\*  
 \* Procedure to delete the selected Drawable from the network  
 \* Removes the Drawable from the DrawableList  
 \* If the Drawable is a node, it is removed from the NodeList, and the directly connected Tasks are also removed  
 \*/* **void** deleteSelected(){  
 **if** (**selectedDrawable**!= **null**){  
 **if** (**selectedDrawable**.getObjectPriority() == 2){  
 **N\_LIST**.remove((Node) **selectedDrawable**);  
 **D\_LIST**.deleteConnected((Node) **selectedDrawable**);  
 }  
 **D\_LIST**.remove(**selectedDrawable**);  
 **selectedDrawable**.delete();  
 }  
 System.***out***.println(**"Selected Deleted"**);  
 repaint();  
 }  
 */\*\*  
 \* Procedure to set the selected drawable to null  
 \*/* **void** selectNull(){  
 **if** (**selectedDrawable**!=**null**) {  
 **selectedDrawable**.setSelected(**false**);  
 repaint();  
 }  
 }  
  
 */\*\*  
 \* Procedure to process a ctrlClick  
 \* If a TaskVisualised is selected, it is toggled to be identified as critical or not critical  
 \** ***@param v*** *- The vector location of the mouse click  
 \*/* **void** ctrlClick (Vector v) {  
 **if** (**selectedDrawable**!= **null**) {  
 **selectedDrawable**.setSelected(**false**);  
 }  
 **if** (**D\_LIST**.getFirst() != **null**) {  
 **selectedDrawable** = **D\_LIST**.whichDrawableContains(v);  
 **if** (**selectedDrawable**!= **null**) {  
 **if** (**selectedDrawable**.getObjectPriority() == 1) { *//Safe to cast because selectedDrawable is proven to be a TaskVisualised* TaskVisualised tVis = (TaskVisualised) **selectedDrawable**;  
 tVis.toggleCritSelected();  
 }  
 }  
 }  
 repaint();  
  
 }  
  
 */\*\*  
 \* Procedure to reset the Component  
 \*/* **void** clearComponent(){  
 **D\_LIST**.removeAll();  
 **N\_LIST**.removeAll();  
 **currentIndex** = 0;  
 repaint();  
 }  
  
 */\*\*  
 \* Function to get the NodeList  
 \** ***@return*** *- The NodeList  
 \*/* NodeList getN\_LIST() {  
 **return N\_LIST**;  
 }  
  
 */\*\*  
 \* Function to get the DrawableList  
 \** ***@return*** *- The DrawableList  
 \*/* DrawableList getD\_LIST(){  
 **return D\_LIST**;  
 }  
}

Drawable (Interface)

**import** java.awt.\*;  
  
*/\*\*  
 \* Interface for all Drawable objects  
 \* Contains abstract methods, which are needed for the DrawableList  
 \*/***public interface** Drawable {  
 **void** draw(Graphics g);  
  
 Drawable getNextDrawable();  
  
 **void** setNextDrawable(Drawable d);  
  
 **int** getObjectPriority();  
  
 **boolean** priorityGreaterThanOrEqual(Drawable d);  
  
 **void** setSelected(**boolean** selected);  
  
 **boolean** checkIfInside(Vector v);  
  
 **void** delete();  
  
 **void** setDisplayingAnswers(**boolean** b);  
}

Node

**import** javax.swing.\*;  
**import** java.awt.\*;  
**import** java.util.HashSet;  
  
*/\*\*  
 \* The Class for the Node object  
 \* ImplementsA the Drawable interface, since it is a drawable object  
 \* The node object displays to the user their inputted early and late event times  
 \* It also links Tasks together  
 \* Each node has a unique index and HashSets of preceding and succeeding tasks  
 \*/***public class** Node **implements** Drawable {  
 **private final int INDEX**;  
 **private int earlyStartInput**, **latestStartInput**, **calculatedEarlyStart**, **calculatedLatestStart**;  
 **private boolean selected**;  
 **private** Vector **centre**;  
 **private** Drawable **nextDraw**;  
 **private** Node **nextNode**;  
 **private** HashSet<Task> **succeedingTasks**;  
 **private** HashSet<Task> **precedingTasks**;  
 **private int selectedArea**;  
 **private boolean checkingAns**, **displayingAnswers**;  
  
 */\*\*  
 \* Constructor for the node class  
 \* Sets the centre location and the index of the node  
 \* Initialises the Task HashSets  
 \* Sets the node as selected  
 \** ***@param index*** *- The unique identifier for the node  
 \** ***@param centre*** *- The Vector location of the centre of the node  
 \*/* Node(**int** index, Vector centre){  
 **this**.**INDEX** = index;  
 **this**.**centre** = centre;  
 **precedingTasks** = **new** HashSet<>();  
 **succeedingTasks** = **new** HashSet<>();  
 **this**.**selected** = **true**;  
 **checkingAns** = **false**;  
 }  
  
 */\*\*  
 \* Procedure to delete the node  
 \* Implementation of the abstract function in the Drawable interface  
 \*/* **public void** delete(){  
 **succeedingTasks** = **null**;  
 **precedingTasks** = **null**;  
 }  
 */\*\*  
 \* Procedure to display the node and all its information onto a path component  
 \* Implementation of the abstract function in the Drawable interface  
 \*  
 \* Fills the main circle of the node with the colour of the panel background,  
 \* This hides the task ends  
 \* If the user has chosen to check their answers, correct input boxes are filled in green,  
 \* incorrect input boxes are filled in red  
 \*  
 \* Then if the node is selected, the edges for the node and the event time box are filled in  
 \* with a different shade of red  
 \* If the node is not selected the edges are coloured black  
 \* the index is displayed in the middle of the circle  
 \* It is coloured red if selected, black if not  
 \* The event times displayed in the boxes are always in the centre of the boxes,  
 \* this is done by calculating the centre of the text based on how many characters it is#  
 \* If displaying calculated event times, the event times are written in blue  
 \** ***@param g*** *- Graphics object  
 \*/* **public void** draw(Graphics g){  
 g.setColor(UIManager.*getColor* ( **"Panel.background"** ));  
 g.fillOval(**centre**.getIntX() - 20,**centre**.getIntY() - 20,40,40);  
 **if** (**checkingAns**){  
 **if** (**earlyStartInput** == **calculatedEarlyStart**){  
 g.setColor(**new** Color(23, 135, 5));  
 }  
 **else** {  
 g.setColor(**new** Color(165, 5, 15));  
 }  
 g.fillRect(**centre**.getIntX() - 30, **centre**.getIntY() - 50, 30, 25);  
  
 **if** (**latestStartInput** == **calculatedLatestStart**){  
 g.setColor(**new** Color(23, 135, 5));  
 }  
 **else** {  
 g.setColor(**new** Color(165, 5, 15));  
 }  
 g.fillRect(**centre**.getIntX(), **centre**.getIntY() - 50, 30, 25);  
 }  
 **else** {  
 g.fillRect(**centre**.getIntX() - 30, **centre**.getIntY() - 50, 30, 25);  
 g.fillRect(**centre**.getIntX(), **centre**.getIntY() - 50, 30, 25);  
 }  
 **if** (**selected**){  
 g.setColor(Color.***RED***);  
 }  
 **else**{  
 g.setColor(Color.***BLACK***);  
 }  
 g.drawOval(**centre**.getIntX() - 20,**centre**.getIntY() - 20,40,40);  
 g.drawRect(**centre**.getIntX() - 30, **centre**.getIntY() - 50, 30, 25);  
 g.drawRect(**centre**.getIntX(), **centre**.getIntY() - 50, 30, 25);  
 g.drawString(Integer.*toString*(**INDEX**), **centre**.getIntX() - ((Integer.*toString*(**INDEX**).length())\*g.getFont().getSize()/3), **centre**.getIntY() + g.getFont().getSize()/3);  
 **if** (**checkingAns**) {  
 g.setColor(Color.***BLACK***);  
 }  
 **if** ((**displayingAnswers**)){  
 g.setColor(Color.***BLUE***);  
 g.drawString(Integer.*toString*(**calculatedEarlyStart**), (**centre**.getIntX() - 15) - ((Integer.*toString*(**calculatedEarlyStart**).length())\*g.getFont().getSize()/3), (**centre**.getIntY() - 37) + g.getFont().getSize()/3);  
 g.drawString(Integer.*toString*(**calculatedLatestStart**), (**centre**.getIntX() + 15) - ((Integer.*toString*(**calculatedLatestStart**).length())\*g.getFont().getSize()/3), (**centre**.getIntY() - 37) + g.getFont().getSize()/3);  
 }  
 **else** {  
 g.drawString(Integer.*toString*(**earlyStartInput**), (**centre**.getIntX() - 15) - ((Integer.*toString*(**earlyStartInput**).length()) \* g.getFont().getSize() / 3), (**centre**.getIntY() - 37) + g.getFont().getSize() / 3);  
 g.drawString(Integer.*toString*(**latestStartInput**), (**centre**.getIntX() + 15) - ((Integer.*toString*(**latestStartInput**).length()) \* g.getFont().getSize() / 3), (**centre**.getIntY() - 37) + g.getFont().getSize() / 3);  
 }  
  
 }  
  
 */\*\*  
 \* Function to check if a click location is within the node  
 \* Implementation of the abstract function in the Drawable interface  
 \*  
 \* Uses vector mathematics to find the vector from the centre of the node to the click  
 \* Checks if the magnitude of this vector is shorter or equal to the length of the radius  
 \* If this is true then the click is within the circle, so returns true and sets selected area to the main circle  
 \* Else, checks if the click is within one of the event time boxes  
 \* Returns true and sets the selected area to the corresponding box if this is true  
 \* Else returns false  
 \** ***@param v*** *- Click Location  
 \** ***@return*** *- If the node is inside the vector  
 \*/* **public boolean** checkIfInside(Vector v){  
 Vector cToT = v.subtract(**centre**);  
 **if** (cToT.getMagnitude()<= 20){ *//If vector from centre to mouse click has magnitude <= 20* **selectedArea** = 1;  
 **return true**;  
 }  
 **else if** ((v.getIntX() >= **centre**.getIntX() - 30) && (v.getIntX() <= **centre**.getIntX() + 30)){  
 **if** ((v.getIntY() >= **centre**.getIntY() - 50 ) && (v.getIntY() <= **centre**.getIntY() - 25)){  
 **if** (v.getIntX() >= **centre**.getIntX()){  
 **selectedArea** = 3; *//Clicked on latest Start area* **return true**;  
 }  
 **selectedArea** = 2; *//Clicked on earliest start area* **return true**;  
 }  
 }  
 **return false**; *//Not clicked on any node area* }  
  
 */\*\*  
 \* Procedure to move a node to a colliding node at the closest point to the mouse  
 \* Finds the joining unit vector between the click and colliding node  
 \* Multiplies this vector by the diameter to find the new node centre location which would place the node at the edge of the colliding node  
 \** ***@param v*** *- Mouse location  
 \** ***@param collidingNode*** *- Node which the mouse is inside  
 \*/* Vector moveToNode(Vector v,Node collidingNode){  
 Vector v2 = collidingNode.getCentre();  
 Vector vToV2 = v2.subtract(v);  
 vToV2 = (vToV2.getUnitVector()).multiply(40);  
 **return** v2.subtract(vToV2);  
 }  
  
 */\*\*  
 \* Procedure to set whether or not the node is displaying the calculated answers  
 \** ***@param b*** *- new value for displayingAnswers  
 \*/* **public void** setDisplayingAnswers(**boolean** b){  
 **if** (b) {  
 **checkingAns** = **false**;  
 }  
 **displayingAnswers** = b;  
 }  
 */\*\*  
 \* Function to get the node index  
 \** ***@return*** *- Node index  
 \*/* **int** getINDEX() {  
 **return INDEX**;  
 }  
  
 */\*\*  
 \* Function to get the Drawable in the DrawableList which follows this node  
 \* Implementation of the abstract function in the Drawable interface  
 \** ***@return*** *- Next Drawable  
 \*/* **public** Drawable getNextDrawable(){  
 **return nextDraw**;  
 }  
  
 */\*\*  
 \* Function to get the next node in the NodeList  
 \** ***@return*** *- Next Node  
 \*/* Node getNextNode() {  
 **return nextNode**;  
 }  
  
 */\*\*  
 \* Procedure to set a new following Drawable in the DrawableList  
 \* Implementation of the abstract procedure in the Drawable interface  
 \** ***@param nextDraw*** *- Next Drawable  
 \*/* **public void** setNextDrawable(Drawable nextDraw) {  
 **this**.**nextDraw** = nextDraw;  
 }  
  
 */\*\*  
 \* Procedure to set a new following node in the NodeList  
 \** ***@param nextNode*** *- Next Node  
 \*/* **void** setNextNode(Node nextNode) {  
 **this**.**nextNode** = nextNode;  
 }  
  
 */\*\*  
 \* Procedure to add a Task to the HashSet of preceding tasks  
 \** ***@param t*** *- New Task  
 \*/* **void** addToPreceding (Task t){  
 **precedingTasks**.add(t);  
 }  
  
 */\*\*  
 \* Function to get the Preceding task HashSet  
 \** ***@return*** *- Preceding task HashSet  
 \*/* HashSet<Task> getPrecedingTasks() {  
 **return precedingTasks**;  
 }  
  
 */\*\*  
 \* Procedure to add a Task to the HashSet of succeeding tasks  
 \** ***@param t*** *- New Task  
 \*/* **void** addToSucceeding(Task t){  
 **succeedingTasks**.add(t);  
 }  
  
 */\*\*  
 \* Function to get the Succeeding task HashSet  
 \** ***@return*** *- Succeeding task HashSet  
 \*/* HashSet<Task> getSucceedingTasks() {  
 **return succeedingTasks**;  
 }  
  
 */\*\*  
 \* Function to get the Vector location of the centre of the node  
 \** ***@return*** *- Centre location vector  
 \*/* Vector getCentre() {  
 **return centre**;  
 }  
  
 */\*\*  
 \* Function to set the Vector location of the centre of the node  
 \** ***@param v*** *- new location for centre  
 \*/* **void** setCentre(Vector v){  
 **centre** = v;  
 }  
  
 */\*\*  
 \* Function to get the priority of the object for the DrawableList  
 \* Implementation of the abstract function in the Drawable interface  
 \** ***@return*** *- The value 2, as 2 is always the priority of Nodes  
 \*/* **public int** getObjectPriority(){  
 **return** 2;  
 }  
  
 */\*\*  
 \* Function to compare the priority of a Node with the priority of another Drawable  
 \* Implementation of the abstract function in the Drawable interface  
 \** ***@param d*** *- Other Drawable  
 \** ***@return*** *- If the node priority is greater or equal  
 \*/* **public boolean** priorityGreaterThanOrEqual(Drawable d){  
 **return** 2 > d.getObjectPriority();  
 }  
  
 */\*\*  
 \* Procedure to set the node as selected or unselected  
 \** ***@param selected*** *- New value for selected  
 \*/* **public void** setSelected(**boolean** selected) {  
 **this**.**selected** = selected;  
 **if** (!selected){  
 **selectedArea** = -1; *//No area selected* }  
 }  
  
 */\*\*  
 \* Returns integer representing amount of correct inputs  
 \** ***@return*** *- Correct input amount (0 - 2)  
 \*/* **int** getMarksForNode(){  
 **int** marks = 0;  
 **if** (**calculatedEarlyStart** == **earlyStartInput**){  
 marks++;  
 }  
 **if** (**calculatedLatestStart** == **latestStartInput**){  
 marks++;  
 }  
 **return** marks;  
 }  
  
 */\*\*  
 \* Function to get which area of the node is selected  
 \** ***@return*** *- The selected area (-1 if node is not selected)  
 \*/* **int** getSelectedArea(){  
 **return selectedArea**;  
 }  
  
 */\*\*  
 \* Procedure to set the input value for the early start of the node  
 \** ***@param earlyStartInput*** *- The input  
 \*/* **void** setEarlyStartInput(**int** earlyStartInput) {  
 **this**.**earlyStartInput** = earlyStartInput;  
 }  
  
 */\*\*  
 \* Procedure to set the input value for the latest start time of the node  
 \** ***@param latestStartInput*** *- The input  
 \*/* **void** setLatestStartInput(**int** latestStartInput) {  
 **this**.**latestStartInput** = latestStartInput;  
 }  
  
 */\*\*  
 \* Function to get the calculated value for the early start time of the node  
 \** ***@return*** *- The calculated value  
 \*/* **int** getCalculatedEarlyStart() {  
 **return calculatedEarlyStart**;  
 }  
  
 */\*\*  
 \* Procedure to set the calculated value for the early start time of the node  
 \** ***@param calculatedEarlyStart*** *- The calculated value  
 \*/* **void** setCalculatedEarlyStart(**int** calculatedEarlyStart){  
 **this**.**calculatedEarlyStart** = calculatedEarlyStart;  
 }  
  
 */\*\*  
 \* Function to get the calculated value for the latest start time of the node  
 \** ***@return*** *- The calculated value  
 \*/* **int** getCalculatedLatestStart() {  
 **return calculatedLatestStart**;  
 }  
  
 */\*\*  
 \* Procedure to set the calculated value for the latest start time of the node  
 \** ***@param calculatedLatestStart*** *- The calculated value  
 \*/* **void** setCalculatedLatestStart(**int** calculatedLatestStart){  
 **this**.**calculatedLatestStart** = calculatedLatestStart;  
 }  
  
 */\*\*  
 \* Function to see if the node is displaying feedback on inputs  
 \** ***@return*** *- If the answers have been checked  
 \*/* **boolean** isCheckingAns(){  
 **return checkingAns**;  
 }  
  
 */\*\*  
 \* Returns true if the node is connected to a network  
 \** ***@return*** *- If connected  
 \*/* **boolean** isConnected(){  
 **return** !(**succeedingTasks**.isEmpty() && **precedingTasks**.isEmpty());  
 }  
  
 */\*\*  
 \* Procedure to set the node to check answers or to not check answers  
 \** ***@param checkingAns*** *- Boolean to say whether to check answers  
 \*/* **void** setCheckingAns(**boolean** checkingAns) {  
 **this**.**checkingAns** = checkingAns;  
 }  
}

Task

*/\*\*  
 \* Object for the tasks  
 \* Stores all the information calculated by the path functions  
 \* Each task is created in terms of a TaskVisualised, with there being exactly one TaskVisualised to represent each task  
 \*/***class** Task {  
 **private int duration**;  
 **private int criticalCost**;  
 **private int earlyStart**;  
 **private int earlyFinish**;  
 **private int latestStart**;  
 **private final** TaskVisualised **TASK\_VISUALISED**;  
 */\*\*  
 \* Constructor for the task object  
 \* Sets the TASK\_VISUALISED, the duration of the task, and the tasks which are directly dependant on the new task  
 \** ***@param tVis*** *- The TaskVisualised which represents the Task  
 \*/* Task(TaskVisualised tVis){  
 **this**.**TASK\_VISUALISED** = tVis;  
 **this**.**duration** = tVis.getDuration();  
 }  
  
 */\*\*  
 \* Procedure to change the latestStart time of a task  
 \** ***@param cpLength*** *- The critical path length  
 \*/* **void** setLatestStart(**int** cpLength) {  
 **latestStart** = cpLength - **criticalCost**;  
 }  
  
 */\*\*  
 \* Procedure to update the duration of the task, used when information about the TaskVisualiseds may have been changed  
 \* If a task is marked as deleted, it is now disconnected from the nodes  
 \*/* **void** updateDuration(){  
 **this**.**duration** = **TASK\_VISUALISED**.getDuration();  
 **if** (**duration** == -1){  
 **try** {  
 **TASK\_VISUALISED**.getSTART\_NODE().getSucceedingTasks().remove(**this**);  
 }  
 **catch** (Exception startNodeDeleted){  
 *//Start node is deleted, no action needed* }  
 **try** {  
 **TASK\_VISUALISED**.getEND\_NODE().getPrecedingTasks().remove(**this**);  
 }  
 **catch** (Exception endNodeDeleted){  
 *//End node is deleted, no action needed* }  
 }  
 }  
  
 */\*\*  
 \* Function to get the task's critical cost  
 \** ***@return*** *- Critical cost  
 \*/* **int** getCriticalCost() {  
 **return criticalCost**;  
 }  
  
 */\*\*  
 \* Procedure to set the critical cost of the task  
 \** ***@param criticalCost*** *- New value for critical cost  
 \*/* **void** setCriticalCost(**int** criticalCost) {  
 **this**.**criticalCost** = criticalCost;  
 }  
  
 */\*\*  
 \* Function to get the task's early start time  
 \** ***@return*** *- Early start time  
 \*/* **int** getEarlyStart() {  
 **return earlyStart**;  
 }  
  
 */\*\*  
 \* Procedure to set the early start time of the task  
 \** ***@param earlyStart*** *- New value for early start  
 \*/* **void** setEarlyStart(**int** earlyStart) {  
 **this**.**earlyStart** = earlyStart;  
 }  
  
 */\*\*  
 \* Function to get the task's early finish  
 \** ***@return*** *- Early finish time of task  
 \*/* **int** getEarlyFinish() {  
 **return earlyFinish**;  
 }  
  
 */\*\*  
 \* Procedure to set the early finish time of the task  
 \** ***@param earlyFinish*** *- New value for early finish  
 \*/* **void** setEarlyFinish(**int** earlyFinish) {  
 **this**.**earlyFinish** = earlyFinish;  
 }  
  
 */\*\*  
 \* Function to get the task's latest start time  
 \** ***@return*** *- Latest start time of task  
 \*/* **int** getLatestStart() {  
 **return latestStart**;  
 }  
  
 */\*\*  
 \* Function to get the duration of the task  
 \** ***@return*** *- Task's duration  
 \*/* **int** getDuration(){  
 **return this**.**duration**;  
 }  
  
 */\*\*  
 \* Function to get the task's TaskVisualised  
 \** ***@return*** *- The TaskVisualised which represents the task  
 \*/* TaskVisualised getTASK\_VISUALISED() {  
 **return TASK\_VISUALISED**;  
 }  
}

TaskVisualised

**import** java.awt.\*;  
  
**import static** java.awt.Color.***BLACK***;  
**import static** java.awt.Color.***BLUE***;  
  
*/\*\*  
 \* Class for the TaskVisualised object  
 \* Implements the Drawable interface, since it is a drawable object  
 \* Each TaskVisualised represents a task, and is painted into the PathComponent  
 \* Painted as a line with a number to represent its duration, and an arrow to show its direction  
 \* (In comments, 'the Task' refers to the Task which this object represents)  
 \*/***public class** TaskVisualised **implements** Drawable{  
 **private int duration**;  
 **private final** Node **START\_NODE**, **END\_NODE**;  
 **private** Drawable **nextDraw**;  
 **private boolean selected**, **critSelected**, **displayingAnswers**;  
  
 */\*\*  
 \* Constructor for TaskVisualised object  
 \* Defined in terms of a start and end node to help with linking the network  
 \* Duration of task is set to 1 as default  
 \* The TaskVisualised is set as selected  
 \** ***@param nStart*** *- Node where the Task begins  
 \** ***@param nFinish*** *- Node where the Task finishes  
 \*/* TaskVisualised (Node nStart, Node nFinish){  
 **this**.**START\_NODE** = nStart;  
 **this**.**END\_NODE** = nFinish;  
 **this**.**duration** = 1;  
 **this**.**selected** = **true**;  
 **displayingAnswers** = **false**;  
 **nextDraw** = **null**;  
 }  
  
 */\*\*  
 \* Procedure to delete the TaskVisualised  
 \* Sets its duration to -1, marking it as deleted  
 \* Implementation of the abstract function in the Drawable interface  
 \*/* **public void** delete(){  
 **duration** = -1;  
 }  
  
 */\*\*  
 \* Procedure to draw the TaskVisualised onto a component  
 \* Implementation of the abstract procedure in the Drawable interface  
 \* The colour of the line and duration depends on certain conditions:  
 \* If the task is bright red, it is selected  
 \* If the task is dark red, it has either been marked as critical when it shouldn't have,  
 \* or not been marked as critical when it should have been. In both scenarios the user must have selected to check answers  
 \* If the task is blue, it has been marked as critical,  
 \* if the user has chosen to check answers, a blue task shows a task which has been correctly marked as critical  
 \* A dummy task (task with zero duration), will be drawn as a dashed line  
 \** ***@param g*** *- Graphics object  
 \*/* **public void** draw (Graphics g){  
 Graphics2D g2d = (Graphics2D) g;  
 Stroke defaultStroke = g2d.getStroke();  
 **if** (**selected**){  
 g.setColor(Color.***RED***);  
 }  
 **else if** (**duration** == 0){  
 g.setColor(***BLACK***);  
 }  
 **else if** (**displayingAnswers**){  
 **if** (((**END\_NODE**.getCalculatedLatestStart() - **START\_NODE**.getCalculatedEarlyStart()) - **duration**) == 0){  
 g.setColor(***BLUE***);  
 }  
 **else**{  
 g.setColor(***BLACK***);  
 }  
 }  
 **else if** (**critSelected**){  
 g.setColor(***BLUE***);  
 **if** (**END\_NODE**.isCheckingAns()){  
 **if** (((**END\_NODE**.getCalculatedLatestStart() - **START\_NODE**.getCalculatedEarlyStart()) - **duration**) != 0) { *//Not critical but marked as critical* g.setColor(**new** Color(165, 5, 15));  
 }  
 }  
 }  
 **else if** (**END\_NODE**.isCheckingAns()) {  
 **if** (((**END\_NODE**.getCalculatedLatestStart() - **START\_NODE**.getCalculatedEarlyStart()) - **duration**) == 0) { *//Critical but not marked as critical* g.setColor(**new** Color(165, 5, 15));  
 } **else** {  
 g.setColor(Color.***BLACK***);  
 }  
 }  
 **else**{  
 g.setColor(Color.***BLACK***);  
 }  
 g2d.draw(**START\_NODE**.getCentre().getIntersectingLine(**END\_NODE**.getCentre(), 0.7, 20, 30)); *//Draws arrow* g2d.draw(**START\_NODE**.getCentre().getIntersectingLine(**END\_NODE**.getCentre(), 0.7, 20, -30));  
 Vector textLocation = **START\_NODE**.getCentre().getPerpendicularLineEnd(**END\_NODE**.getCentre(), 0.5, 5);  
 **if** (**duration** == 0){ *//Task is a dummy* Stroke dashed = **new** BasicStroke(1, BasicStroke.***CAP\_BUTT***, BasicStroke.***JOIN\_BEVEL***, 0, **new float**[]{9}, 0);  
 g2d.setStroke(dashed);  
 }  
 **else**{  
 g2d.drawString(**""** + **this**.**duration**, textLocation.getIntX(), textLocation.getIntY());  
 }  
 g2d.drawLine((**START\_NODE**.getCentre()).getIntX(), (**START\_NODE**.getCentre()).getIntY(), (**END\_NODE**.getCentre()).getIntX(), (**END\_NODE**.getCentre()).getIntY());  
 g2d.setStroke(defaultStroke);  
 }  
  
 */\*\*  
 \* Function to check if a vector is close enough to a task to be considered 'colliding'  
 \* Implementation of the abstract function in the Drawable interface  
 \** ***@param v*** *- Vector to be compared  
 \** ***@return*** *- If the vector is colliding  
 \*/* **public boolean** checkIfInside(Vector v){  
 **return** v.perpendicularDistanceToLineSeg(**this**.**START\_NODE**.getCentre(), **this**.**END\_NODE**.getCentre()) < 6;  
 }  
  
 */\*\*  
 \* Function to get the Drawable in the DrawableList which follows this TaskVisualised  
 \* Implementation of the abstract function in the Drawable interface  
 \** ***@return*** *- Next drawable  
 \*/* **public** Drawable getNextDrawable() {  
 **return nextDraw**;  
 }  
  
 */\*\*  
 \* Procedure to set a Drawable to follow this TaskVisualised in the DrawableList  
 \* Implementation of the abstract procedure in the Drawable interface  
 \** ***@param nextDraw*** *- New Drawable to be set as next  
 \*/* **public void** setNextDrawable(Drawable nextDraw) {  
 **this**.**nextDraw** = nextDraw;  
 }  
  
 */\*\*  
 \* Procedure to get the duration of the Task  
 \** ***@return*** *- Duration  
 \*/* **int** getDuration() {  
 **return duration**;  
 }  
  
 */\*\*  
 \* Function to get the node which is at the start of the TaskVisualised  
 \** ***@return*** *- The starting node  
 \*/* Node getSTART\_NODE(){  
 **return START\_NODE**;  
 }  
  
 */\*\*  
 \* Function to get the node which is at the end of the TaskVisualised  
 \** ***@return*** *- The finishing node  
 \*/* Node getEND\_NODE(){  
 **return END\_NODE**;  
 }  
  
 */\*\*  
 \* Procedure to set whether the TaskVisualised is selected  
 \** ***@param selected*** *- New value for selected  
 \*/* **public void** setSelected (**boolean** selected){  
 **this**.**selected** = selected;  
 }  
  
 */\*\*  
 \* Procedure to set the duration of the task  
 \* Verifies if new duration is valid (Non negative) before setting duration  
 \** ***@param d*** *- New value for duration  
 \*/* **void** setDuration (**int** d){  
 **if** (**duration** >= 0) {  
 **this**.**duration** = d;  
 }  
 **else**{  
 System.***out***.print(**"Invalid time"**); *//Change to something on gui* }  
 }  
  
 */\*\*  
 \* Procedure to set whether or not the node is displaying the calculated answers  
 \** ***@param b*** *- new value for displayingAnswers  
 \*/* **public void** setDisplayingAnswers (**boolean** b){  
 **displayingAnswers** = b;  
 }  
  
 */\*\*  
 \* Returns 1 if a task is marked correctly  
 \* 0 Otherwise  
 \** ***@return*** *- Amount of marks  
 \*/* **int** checkMark(){  
 **if** (**critSelected** && (((**END\_NODE**.getCalculatedLatestStart() - **START\_NODE**.getCalculatedEarlyStart()) - **duration**) == 0)){  
 **return** 1;  
 }  
 **else if** (((**END\_NODE**.getCalculatedLatestStart() - **START\_NODE**.getCalculatedEarlyStart()) - **duration**) != 0){  
 **return** 1;  
 }  
 **return** 0;  
 }  
  
 */\*\*  
 \* Procedure to toggle whether the Task is recognised as critical or not  
 \*/* **void** toggleCritSelected(){  
 **this**.**critSelected** = !**critSelected**;  
 }  
  
 */\*\*  
 \* Function to get the priority of the object for the DrawableList  
 \* Implementation of the abstract function in the Drawable interface  
 \** ***@return*** *- The value 1, as 1 is always the priority of any TaskVisualised  
 \*/* **public int** getObjectPriority(){  
 **return** 1;  
 }  
 */\*\*  
 \* Function to compare the priority of a Node with the priority of another Drawable  
 \* Implementation of the abstract function in the Drawable interface  
 \** ***@param d*** *- Other Drawable  
 \** ***@return*** *- If the node priority is greater or equal  
 \*/* **public boolean** priorityGreaterThanOrEqual(Drawable d){  
 **return** 1 > d.getObjectPriority();  
 }  
}

DrawableList

**import** java.awt.\*;  
  
*/\*\*  
 \* Linked List class, of Drawable objects  
 \* Used to draw the objects into the PathComponent  
 \* Sorted using a priority system to hold every TaskVisualised first  
 \*/***class** DrawableList {  
 **private** Drawable **first** = **null**;  
  
 */\*\*  
 \* Constructor of list for 0 items  
 \*/* DrawableList() {}  
  
 */\*\*  
 \* Checks if list is empty  
 \** ***@return*** *- If empty return true  
 \*/* **private boolean** isEmpty(){  
 **return this**.**first** == **null**;  
 }  
  
 */\*\*  
 \* Insert procedure. When a Drawable is created, it needs to be added to the list  
 \* This procedure is called to complete this task  
 \* The method iterates through the list until it finds the correct priority of item  
 \* Once the correct priority is reached, the new drawable is inserted at this priority level  
 \* The inserted location will always either be the first location in the list or the last  
 \* The next drawable for the items around the inserted drawable are changed accordingly  
 \** ***@param drawable*** *- Item to be inserted  
 \*/* **void** insert (Drawable drawable){  
 Drawable current = **this**.**first**;  
 Drawable previous = **this**.**first**;  
 **if** (**this**.isEmpty()) **this**.**first** = drawable;  
 **else if** (**this**.**first**.priorityGreaterThanOrEqual(drawable))  
 {  
 drawable.setNextDrawable(**this**.**first**);  
 **this**.**first** = drawable;  
 }  
 **else** {  
 **while** (current !=**null** && !(current.priorityGreaterThanOrEqual(drawable)))  
 {  
 previous = current;  
 current = current.getNextDrawable();  
 }  
 drawable.setNextDrawable(current);  
 previous.setNextDrawable(drawable);  
 }  
 }  
  
 */\*\*  
 \* Iterates through the loop to find a given Drawable  
 \* Once found, the drawable which follows this item is set to follow the previous drawable instead  
 \* This process removes the given drawable from the list  
 \** ***@param drawable*** *- The drawable to be removed  
 \*/* **void** remove (Drawable drawable){  
 **if** (!**this**.isEmpty()){  
 Drawable current = **this**.**first**;  
 Drawable previous = **this**.**first**;  
 **if** (drawable == current){  
  
 **this**.**first** = current.getNextDrawable();  
 }  
 **else** {  
 **while** (current != **null**) {  
 **if** (drawable == current) {  
 previous.setNextDrawable(current.getNextDrawable());  
 current = **null**;  
 } **else** {  
 previous = current;  
 current = current.getNextDrawable();  
 }  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* Removed every item in the list  
 \* By removing the first item, all other items are removed  
 \*/* **void** removeAll(){  
 **first** = **null**;  
 }  
  
 */\*\*  
 \* Function to check whether any TaskVisualised has either the same start and end nodes as the inputted TaskVisualised,  
 \* or if it's end node is the same as another's start, and it's start the same as that tasks end  
 \** ***@param tVis*** *- The task to be found  
 \** ***@return*** *- True if a task matches, false if not  
 \*/* **boolean** containsTask (TaskVisualised tVis){  
 **if** (!**this**.isEmpty()){  
 Drawable current = **this**.**first**;  
 **while** ((current != **null**) &&(current.getObjectPriority() ==1)) {  
 **if** (tVis.getSTART\_NODE() == ((TaskVisualised)current).getSTART\_NODE()) {  
 **if** (tVis.getEND\_NODE() == ((TaskVisualised) current).getEND\_NODE()) {  
 **return true**;  
 }  
 }  
 **else if** (tVis.getSTART\_NODE() == ((TaskVisualised)current).getEND\_NODE()) {  
 **if** (tVis.getEND\_NODE() == ((TaskVisualised) current).getSTART\_NODE()) {  
 **return true**;  
 }  
 }  
 current = current.getNextDrawable();  
 }  
 }  
 **return false**;  
 }  
  
 */\*\*  
 \* Procedure to delete all the Tasks directly connected to a certain Node  
 \* Iterates through every TaskVisualised, and checks if it starts or ends at the given node  
 \* If it does, then it is deleted from the DrawableList, and gets its duration set to -1  
 \* Its duration is set to -1 to show that it has been deleted  
 \** ***@param n*** *- The given Node  
 \*/* **void** deleteConnected (Node n){  
 **if** (!**this**.isEmpty()){  
 Drawable current = **this**.**first**;  
 **if** (current.getObjectPriority() == 1) {  
 **if** ((n == ((TaskVisualised) current).getEND\_NODE()) || (n == ((TaskVisualised) current).getSTART\_NODE())) {  
 remove(current);  
 }  
 **while** ((current != **null**) && (current.getObjectPriority() == 1)) {  
 **if** ((n == ((TaskVisualised) current).getEND\_NODE()) || (n == ((TaskVisualised) current).getSTART\_NODE())) {  
 remove(current);  
 current.delete();  
 }  
 current = current.getNextDrawable();  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* Procedure used to draw objects on screen  
 \* Iterates through the list and runs the draw procedure for every item in list  
 \** ***@param g*** *- Graphics object  
 \*/* **void** drawAll(Graphics g){  
 Drawable current = **this**.**first**;  
 **while** (current != **null**){  
 current.draw(g);  
 current = current.getNextDrawable();  
 }  
 }  
  
 */\*\*  
 \* Function to get the amount of marks awarded for highlighting tasks correctly  
 \* Sums the amount of tasks marked correctly  
 \* Divides this by the amount of tasks  
 \* Multiplies by 2 and rounds down to nearest integer (As tasks are marked out of two)  
 \* Note that dummy tasks are ignored by this algorithm  
 \** ***@return*** *- Rounded mark  
 \*/* **int** checkTaskMarks(){  
 **int** totalMark = 0;  
 **int** amountOfTasks = 0;  
 **if** (!**this**.isEmpty()) {  
 Drawable current = **this**.**first**;  
 **while** ((current != **null**) && (current.getObjectPriority() == 1)) {  
 TaskVisualised tV = (TaskVisualised) current;  
 **if** (tV.getDuration() != 0) {  
 amountOfTasks++;  
 **if** (tV.checkMark() == 1){  
 totalMark++;  
 }  
 }  
  
 current = current.getNextDrawable();  
 }  
 }  
 **return** (2 \* totalMark) / (amountOfTasks);  
 }  
 */\*\*  
 \* Function to find if a click location is within the area of an object  
 \* Iterates through the nodes first, running the checkIfInside function for each node  
 \* Then iterates through the TaskVisualised objects, and runs their checkIfInside functions  
 \** ***@param v*** *- Click Vector location  
 \** ***@return*** *- Which drawable contains the vector, returns null if none contain  
 \*/* Drawable whichDrawableContains(Vector v) {  
 Drawable current = **this**.**first**;  
 **while** (current != **null**) {  
 **if** (current.getObjectPriority() == 2) { *//Check nodes first* **if** (current.checkIfInside(v)) {  
 **return** current;  
 }  
 }  
 current = current.getNextDrawable();  
 }  
 current = **this**.**first**;  
 **while** (current.getObjectPriority() == 1){  
 **if** (current.checkIfInside(v)) {  
 **return** current;  
 }  
 current = current.getNextDrawable();  
 }  
 **return null**;  
 }  
  
 */\*\*  
 \* Procedure to iterate through all Drawables and make them display the calculated answers/if critical  
 \** ***@param b*** *- True if answers should be shown, false if they should stop being shown  
 \*/* **void** displayAnswers(**boolean** b){  
 Drawable current = **this**.**first**;  
 **while** (current != **null**){  
 current.setDisplayingAnswers(b);  
 current = current.getNextDrawable();  
 }  
 }  
  
 */\*\*  
 \* Function to get the first item in list  
 \** ***@return*** *- First  
 \*/* Drawable getFirst() {  
 **return first**;  
 }  
}

NodeList

**import** java.util.Iterator;  
  
*/\*\*  
 \* Linked List class, of Nodes  
 \* Used to perform checks on all nodes and to display whether the user has inputted the correct event times  
 \*/***class** NodeList {  
 **private** Node **first** = **null**;  
  
 */\*\*  
 \* Empty constructor for list with zero items  
 \*/* NodeList() {  
 }  
  
 */\*\*  
 \* Function to get whether the list is empty  
 \** ***@return*** *- Whether list is empty  
 \*/* **private boolean** isEmpty() {  
 **return this**.**first** == **null**;  
 }  
  
 */\*\*  
 \* Function to check whether a vector location of the centre of a node lies within the distance of one radius away from any of the nodes in the list  
 \* Uses a while loop to iterate through the list and check if the joining vector between the two centres is  
 \* within the range of 0 to the sum of the radii  
 \* If true then the nodes intersect, the intersecting node is returned  
 \** ***@param v2*** *- The location of the vector  
 \** ***@param exclude*** *- The node which is excluded from the search (Used to stop a node being dragged from intersecting with itself)  
 \** ***@return*** *- The intersecting node  
 \*/* Node circleIntersectCheck(Vector v2, **int** exclude) {  
 Node current = **this**.**first**;  
 **while** (current != **null**) {  
 **if** (current.getINDEX() != exclude) {  
 **double** magSquared = current.getCentre().getJoiningVectorMagnitudeSquared(v2);  
 **if** (magSquared <= 1600) { *// (R0 - R1)^2 <= (x0 - x1)^2 + (y0 - y1)^2 <= (R0 + R1)^2 Equation squared to avoid sqrt() function for efficiency  
 // System.out.println(magSquared);* **return** current;  
 }  
 }  
 current = current.getNextNode();  
 }  
 **return null**;  
 }  
  
 */\*\*  
 \* Iterates through the loop to find a given node  
 \* Once found, the node which follows this item is set to follow the previous node instead  
 \* This process removes the given node from the list  
 \** ***@param n*** *- The drawable to be removed  
 \*/* **void** remove (Node n){  
 **if** (!**this**.isEmpty()){  
 Node current = **this**.**first**;  
 Node previous = **this**.**first**;  
 **if** (n == current){  
 **this**.**first** = current.getNextNode();  
 }  
 **else** {  
 **while** (current != **null**) {  
 **if** (n == current) {  
 previous.setNextNode(current.getNextNode());  
 current = **null**;  
 } **else** {  
 previous = current;  
 current = current.getNextNode();  
 }  
 }  
 }  
 }  
 }  
  
 */\*\*  
 \* Removed every item in the list  
 \* By removing the first item, all other items are removed  
 \*/* **void** removeAll(){  
 **first** = **null**;  
 }  
  
 */\*\*  
 \* Function to return which node contains a vector location  
 \* Iterates through the list until a node is found to contain the click location  
 \* If no node contains the click then return null  
 \** ***@param vT*** *- Click location  
 \** ***@return*** *- The intersecting node  
 \*/* Node whichNodeContains(Vector vT) {  
 Node current = **this**.**first**;  
 **while** (current != **null**) {  
 **if** (current.checkIfInside(vT)) {  
 **return** current;  
 }  
 current = current.getNextNode();  
 }  
 **return null**;  
 }  
  
 */\*\*  
 \* Procedure to insert a node into the list  
 \* Iterates through list to find the last node in the list  
 \* Sets the last nodes 'next' to the new node  
 \** ***@param item*** *- New Node to be added  
 \*/* **void** insert(Node item) {  
 Node current = **this**.**first**;  
 Node previous = **this**.**first**;  
 **if** (**this**.isEmpty()) **this**.**first** = item;  
 **else** {  
 **while** (current != **null**) {  
 previous = current;  
 current = current.getNextNode();  
 }  
 previous.setNextNode(item);  
 }  
 }  
  
 */\*\*  
 \* Procedure to check the inputted answers for every node  
 \* Iterates through each node, setting the node into 'checkingAns' mode  
 \* If a node is at the end of the network, its times are evaluated  
 \*  
 \* nodeCount = amount of nodes in the path, used to get fraction of inputs which are correct,  
 \* This fraction is multiplied by 4 and rounded down to get total marks for nodes  
 \*/* **int** checkAnswers(){  
 Node current = **this**.**first**;  
 **int** nodeCount = 0;  
 **int** marks = 0;  
 **while** (current != **null**){  
 Iterator<Task> it = current.getSucceedingTasks().iterator();  
 **if** (it.hasNext()) {  
 Task t = it.next();  
 current.setCalculatedEarlyStart(t.getEarlyStart());  
 current.setCalculatedLatestStart(t.getLatestStart());  
 current.setCheckingAns(**true**);  
 marks += current.getMarksForNode();  
 nodeCount++;  
 }  
 **else** { *//Last node in network* System.***out***.println(current.getINDEX());  
 Iterator<Task> i = current.getPrecedingTasks().iterator();  
 **if** (i.hasNext()) {  
 Task t = i.next();  
 current.setCalculatedEarlyStart(t.getEarlyFinish());  
 current.setCalculatedLatestStart(t.getEarlyFinish());  
 current.setCheckingAns(**true**);  
 marks += current.getMarksForNode();  
 nodeCount++;  
 }  
 }  
 current = current.getNextNode();  
 }  
 **int** totalMarks;  
 **try** {  
 totalMarks = (4 \* marks) / (2 \* nodeCount);  
 }  
 **catch** (Exception zeroNodes){ *//If there is no nodes, the above expression = n/0, which is undefined for all values of n* totalMarks = 0;  
 }  
 System.***out***.println(totalMarks);  
 **return** totalMarks;  
 }  
  
 */\*\*  
 \* Procedure to iterate through all the nodes and stop them from checking inputted answers  
 \* Removes all calculated times  
 \*/* **void** setCheckAnswersFalse(){  
 Node current = **this**.**first**;  
 **while** (current != **null**){  
 System.***out***.println(current.getINDEX());  
 current.setCheckingAns(**false**);  
 current.setCalculatedEarlyStart(0);  
 current.setCalculatedLatestStart(0);  
 current = current.getNextNode();  
 }  
 }  
}

Vector

**import** java.awt.geom.Line2D;  
  
*/\*\*  
 \* Class for Vector object  
 \* a 'Vector', is a 2 dimensional object, consisting of two doubles  
 \* Each double represents a component of x or y, for example a Vector (4, 3) would mean 4 in the x direction, and 3 in the y direction  
 \* Any location on a 2d plane can be represented using a Vector  
 \*/***class** Vector {  
 **private double x**, **y**;  
 Vector(**double** x, **double** y){  
 **this**.**x** = x;  
 **this**.**y** = y;  
 }  
  
 */\*\*  
 \* Function to add two Vectors together  
 \* Adds the x and y components together, and returns the resultant Vector  
 \** ***@param v2*** *- Vector to be added  
 \** ***@return*** *- Resultant vector  
 \*/* **private** Vector add(Vector v2){  
 **double** xA = **this**.**x** + v2.getX();  
 **double** yA = **this**.**y** + v2.getY();  
 **return new** Vector(xA, yA);  
 }  
  
 */\*\*  
 \* Function to subtract one Vector from another  
 \* Subtracts the Vector passed in from the Vector referenced  
 \** ***@param v2*** *- The vector to be subtracted  
 \** ***@return*** *- The resultant Vector  
 \*/* Vector subtract(Vector v2){  
 **double** xA = **this**.**x** - v2.getX();  
 **double** yA = **this**.**y** - v2.getY();  
 **return new** Vector(xA, yA);  
 }  
  
 */\*\*  
 \* Function to get the magnitude of the Vector  
 \* Uses Pythagoras's theorem treating the Vector components as a triangle, with the hypotenuse as the magnitude  
 \** ***@return*** *- Magnitude  
 \*/* **double** getMagnitude(){  
 **return** (Math.*sqrt*((**x**\***x**) + (**y**\***y**)));  
 }  
  
 */\*\*  
 \* Function to get the squared magnitude of the Vector  
 \* Used to avoid 'slow' Math.sqrt() function  
 \** ***@return*** *- Magnitude Squared  
 \*/* **private double** getMagnitudeSquared(){  
 **return** (**x**\***x**) + (**y**\***y**);  
 }  
  
 */\*\*  
 \* Function to get the shortest distance between a point and a line segment  
 \* The closest distance between a point and a line is always equal to the normal from the line which intersects the point  
 \* The function uses vector mathematics to find this normal, and then calculate its magnitude from the line to the point  
 \* It follows the following procedure:  
 \* Finds the vector from the start of the task to the point  
 \* Gets the vector from the start of the segment to the end of the segment  
 \* Verifies that the line has a length greater than 0  
 \* Finds the dot product of the line segment and the vector from the start of the line to the point  
 \* Divides this dot product by the line’s magnitude squared, and store as a decimal  
 \* Multiplies the length of the line by the decimal, and add this to the start of the line, to get the vector from the origin to the closest point on the line  
 \* Subtracts this vector from the vector from the origin to the point, giving the shortest vector from the point to line  
 \* Returns the magnitude of this vector  
 \* Note, the point is the Vector referenced by 'this'  
 \** ***@param start*** *- Start of segment  
 \** ***@param end*** *- End of segment  
 \** ***@return*** *- Magnitude of joining Vector  
 \*/* **double** perpendicularDistanceToLineSeg(Vector start, Vector end){  
 Vector startToPoint = **this**.subtract(start);  
 Vector line = end.subtract(start);  
 **double** p = -1;  
 **if** (line.getMagnitudeSquared() != 0){  
 p = startToPoint.getDotProduct(line) / line.getMagnitudeSquared();  
 }  
 Vector d = **this**.subtract(start.add(line.multiply(p)));  
 **return** d.getMagnitude();  
 }  
  
 */\*\*  
 \* Function to get a line segment, of given length, which intersects another segment, at a given acute angle  
 \* The intersection needs to be a given fraction across the original segment  
 \* It follows this procedure:  
 \* Finds the point on the task where the new line will begin  
 \* Gets the vector joining the start of the node to the task  
 \* Converts the joining vector into polar co-ordinates  
 \* Adds the angle of the polar line to the inputted angle of intersection  
 \* Finds the x coordinate for the end of the new line  
 \* Finds the y coordinate for the end of the new line  
 \* Returns the line joining the start of the line to the end coordinates  
 \* Note, the start of the given segment is referenced by 'this'  
 \** ***@param end*** *- The end of the given segment  
 \** ***@param mult*** *- The fraction of the way across the segment the new line will intersect  
 \** ***@param length*** *- The length of the new segment  
 \** ***@param angle*** *- The acute angle of intersection  
 \** ***@return*** *- The new line  
 \*/* Line2D.Double getIntersectingLine(Vector end, **double** mult, **int** length, **double** angle){  
 Vector point =(end.subtract(**this**)).multiply(mult).add(**this**);  
 Vector startToPoint = point.subtract(**this**);  
 angle = Math.*toRadians*(angle);  
 **double** theta = Math.*atan2*(startToPoint.getY(), startToPoint.getX());  
 **double** x2, y2, rho = theta + angle;  
 x2 = point.getX() - length \* Math.*cos*(rho);  
 y2 = point.getY() - length \* Math.*sin*(rho);  
 **return new** Line2D.Double(point.getX(), point.getY(), x2, y2);  
 }  
  
 */\*\*  
 \* Function to find a point which is a given perpendicular distance away from a given line, at a given point  
 \* Follows a very similar procedure to the 'getIntersectingLine' function, but returns a vector instead of a line  
 \** ***@param end*** *- The end of the given segment  
 \** ***@param mult*** *- The fraction of the way across the new line the normal will intersect  
 \** ***@param length*** *- The length across the normal, where the new vector will be  
 \** ***@return*** *- The location found  
 \*/* Vector getPerpendicularLineEnd(Vector end, **double** mult, **int** length){  
 Vector point =(end.subtract(**this**)).multiply(mult).add(**this**);  
 Vector d = point.subtract(**this**);  
 **double** theta = Math.*atan2*(d.getY(), d.getX());  
 **double** x, y, rho = theta + Math.***PI*** / 2;  
 x = point.getX() - length \* Math.*cos*(rho);  
 y = point.getY() - length \* Math.*sin*(rho);  
 **return new** Vector(x, y);  
 }  
  
 */\*\*  
 \* Finds the squared magnitude of a vector which joins two vector locations  
 \* Uses Pythagoras's theorem  
 \** ***@param v2*** *- Second Vector  
 \** ***@return*** *- Joining magnitude squared  
 \*/* **double** getJoiningVectorMagnitudeSquared (Vector v2){  
 Vector vJoining = v2.subtract(**this**);  
 **return** (vJoining.getX() \* vJoining.getX()) + (vJoining.getY() \* vJoining.getY());  
 }  
  
 */\*\*  
 \* Function to get the unit vector  
 \* The unit vector of a vector is a vector of the same direction, but with a magnitude of 1  
 \** ***@return*** *- The unit vector  
 \*/* Vector getUnitVector(){  
 **return new** Vector(**x**/**this**.getMagnitude(),**y**/**this**.getMagnitude());  
 }  
  
 */\*\*  
 \* Function to multiply a vector by a double  
 \* Multiplies both of the vector's components by the multiple  
 \** ***@param n*** *- Multiple  
 \** ***@return*** *- The multiplied Vector  
 \*/* Vector multiply (**double** n) {  
 **return new** Vector(**x** \* n, **y** \* n);  
 }  
  
 */\*\*  
 \* Function to get the dot product of two vectors  
 \* The dot product is equal to the sum of the components of the two vectors multiplied together  
 \** ***@param v*** *- The second Vector  
 \** ***@return*** *- Dot product of the Vector  
 \*/* **private double** getDotProduct (Vector v){  
 **return** (**x**\*v.getX()) + (**y**\*v.getY());  
 }  
  
 */\*\*  
 \* Function to get the x component of the Vector  
 \** ***@return*** *- x component  
 \*/* **private double** getX() {  
 **return x**;  
 }  
  
 */\*\*  
 \* Function to get the y component of the Vector  
 \** ***@return*** *- y component  
 \*/* **private double** getY() {  
 **return y**;  
 }  
  
 */\*\*  
 \* Function to get the x component of the Vector rounded to the nearest integer  
 \** ***@return*** *- rounded x component  
 \*/* **int** getIntX(){  
 **return** (**int**) Math.*round*(**x**);  
 }  
 */\*\*  
 \* Function to get the y component of the Vector rounded to the nearest integer  
 \** ***@return*** *- rounded y component  
 \*/* **int** getIntY(){  
 **return** (**int**) Math.*round*(**y**);  
 }  
  
 */\*\*  
 \* Procedure to set x  
 \** ***@param x*** *- New value for x  
 \*/* **void** setX(**double** x){  
 **this**.**x** = x;  
 }  
 */\*\*  
 \* Procedure to set y  
 \** ***@param y*** *- New value for y  
 \*/* **void** setY(**double** y){  
 **this**.**y** = y;  
 }  
}

PathFunctions

**import** java.util.HashSet;  
**import** java.util.Iterator;  
**import** java.util.Set;  
  
*/\*\*  
 \* Class with methods to perform algorithms to calculate late event times, early event times, and critical information about tasks  
 \*/***class** PathFunctions {  
 **private final** HashSet<Task> **TASKS** = **new** HashSet<>();  
 **private int cpLength**;  
  
 */\*\*  
 \* Procedure to add a task to the set of tasks which are part of the network  
 \* If null, the task is not added  
 \** ***@param t*** *- The task to be added  
 \*/* **void** addTask(Task t){  
 **if** (t != **null**) {  
 **TASKS**.add(t);  
 }  
 }  
  
 */\*\*  
 \* Procedure to remove all the tasks from the HashSet  
 \*/* **void** deleteAllTasks(){  
 **TASKS**.clear();  
 }  
 */\*\*  
 \* Procedure used to calculate all the information needed about a set of tasks  
 \* Updates all the Task's durations, if a Task has a duration of -1, it has been removed from the network,  
 \* so needs to be removed from the HashSet  
 \* Also sets all the values to be calculated back to 0, ensuring previous calculations do not interfere with the algorithm  
 \* The algorithm starts by creating the ‘completed’ set, which holds the tasks whose critical costs have been calculated  
 \* In addition to the ‘remaining’ set, this holds the rest of the tasks.  
 \* Initially, ‘remaining’ holds all the tasks  
 \* The first ‘for’ and ‘while’ loops iterate through every task still in ‘remaining’, until ‘remaining’ is empty  
 \* On each iteration, the code within the loop tries to calculate the criticalCost for the current task  
 \* To calculate the criticalCost, every task which the current task is dependent on needs to have a calculated criticalCost  
 \* To verify this, an ‘if’ statement checks if all the dependencies are in the completed set  
 \* If they are not, the loop re-iterates with the next task  
 \* However, if the statement returns true, the ‘critical’ variable is set to the largest criticalCost of all the dependencies  
 \* Then the criticalCost of the current task is calculated, and the task can now be marked as completed  
 \* So it is added to the set called 'completed', and removed from 'remaining'  
 \* Progress is marked as true if this happens  
 \* If no progress is made over the whole of the ‘for’ loop, then the path must contain a cycle, which means there is no path from start to finish.  
 \* On this occasion, the user will be notified that the path they have created is cyclic  
 \* Then uses methods to calculate the max cost and early start and finishes of the task  
 \*/* **void** criticalPath(){  
 Task[] markedToDelete = **new** Task[**TASKS**.size()];  
 **int** markedToDeleteIndex = 0;  
 **for** (Task current: **TASKS**){  
 current.updateDuration();  
 current.setEarlyStart(0);  
 current.setEarlyFinish(0);  
 current.setCriticalCost(0);  
 current.setLatestStart(0);  
 **if** (current.getDuration() == -1){  
 markedToDelete[markedToDeleteIndex] = current; *//Need to add to array and delete after, to avoid concurrent modification exception* markedToDeleteIndex++;  
 System.***out***.print(**"DEL"**);  
 }  
 }  
 **for** (Task t : markedToDelete){  
 **TASKS**.remove(t); *//Now the deleted tasks can be safely removed from the HashSet* }  
 HashSet<Task> completed = **new** HashSet<>();  
 HashSet<Task> remaining = **new** HashSet<>(**TASKS**);  
 **while** (!remaining.isEmpty()) {  
 **boolean** progress = **false**;  
 **for** (Iterator<Task> it = remaining.iterator(); it.hasNext();) {  
 Task task = it.next();  
 System.***out***.println(task.getTASK\_VISUALISED().getSTART\_NODE().getINDEX());  
 **if** (completed.containsAll(task.getTASK\_VISUALISED().getEND\_NODE().getSucceedingTasks())) { *// all the tasks dependencies have calculated costs* **int** critical = 0;  
 **for** (Task t : task.getTASK\_VISUALISED().getEND\_NODE().getSucceedingTasks()) { *// Find the greatest critical cost preceding the current task* **if** (t.getCriticalCost() > critical) {  
 critical = t.getCriticalCost();  
 }  
 }  
 task.setCriticalCost(critical + task.getDuration());  
 completed.add(task); *// task now has the greatest value of critical cost it can have* it.remove();  
 progress = **true**;  
 }  
 }  
 **if** (!progress) { *// Cycle exists if no progress* **throw new** RuntimeException(**"Cycle exists in path"**);  
 }  
 }  
 cpLength(); *// calculate cost and early starts* HashSet<Task> initialTasks = initials(**TASKS**);  
 calculateEarly(initialTasks);  
 }  
  
 */\*\*  
 \* Procedure to set the calculate the early event times for initial tasks to 0, and then start setting the following tasks' event times  
 \** ***@param initials*** *- The initial tasks in the network  
 \*/* **private void** calculateEarly(HashSet<Task> initials) {  
 **for** (Task initial : initials) {  
 initial.setEarlyStart(0);  
 initial.getTASK\_VISUALISED().getSTART\_NODE().setCalculatedEarlyStart(0);  
 System.***out***.println(**"Setting initial "** + initial.getDuration());  
 initial.setEarlyFinish(initial.getDuration());  
 setEarly(initial);  
 }  
 }  
  
 */\*\*  
 \* Function to identify the initial tasks in the network  
 \* Creates a HashSet of all the tasks  
 \* Iterates through every task, removing tasks which are dependent on the current task from the HashSet  
 \* The tasks remaining in the HashSet are dependent on nothing, so must be at the start of the network  
 \** ***@param tasks*** *- Every task  
 \** ***@return*** *- The HashSet of initial tasks  
 \*/* **private** HashSet<Task> initials(Set<Task> tasks) {  
 HashSet<Task> remaining = **new** HashSet<>(tasks);  
 **for** (Task t : tasks) {  
 **for** (Task td : t.getTASK\_VISUALISED().getEND\_NODE().getSucceedingTasks()) {  
 remaining.remove(td);  
 }  
 }  
 **return** remaining;  
 }  
  
 */\*\*  
 \* Recursive procedure to set the early start and early finish times of all the non-initial tasks  
 \* Iterates through the tasks which are directly dependent on the task passed into the procedure  
 \* Checks if the task has already had its early start time calculated, and that this early start time needs to be changed  
 \* If not, then uses the task passed in to find the current task's early start and early finish times  
 \* Uses a recursive call with the current task as a parameter, to calculate the early times for the task which immediately follows the current task  
 \* This ensures every unique path on the network is followed, and every task gets calculated  
 \* Note the recursive call doesn't occur if a task already had calculated event times, this stops the procedure from following paths which have already been traversed  
 \*  
 \* This algorithm works because, when it is ran, every initial task has their event times calculated,  
 \* and the first call of the procedure is always using an initial task  
 \** ***@param t1*** *- The task with known early event times, used to calculate following tasks  
 \*/* **private void** setEarly(Task t1) {  
 **int** completionTime = t1.getEarlyFinish();  
 **for** (Task t2 : t1.getTASK\_VISUALISED().getEND\_NODE().getSucceedingTasks()) {  
 **if** (completionTime >= t2.getEarlyStart()) {  
 System.***out***.println(**"Setting early start and finish "** + t1.getDuration());  
 t2.setEarlyStart(completionTime);  
 t2.setEarlyFinish(completionTime + t2.getDuration());  
 setEarly(t2);  
 }  
 }  
 }  
  
 */\*\*  
 \* Procedure to calculate the length of the critical path  
 \* Iterates through the complete list of tasks and finds which task has the greatest criticalCost  
 \* The greatest critical cost is equal to the critical path length  
 \*/* **private void** cpLength() {  
 **int** max = -1;  
 **for** (Task t : **TASKS**) {  
 **if** (t.getCriticalCost() > max)  
 max = t.getCriticalCost();  
 }  
 **cpLength** = max;  
 System.***out***.println(**"Critical path length (cost): "** + **cpLength**);  
 **for** (Task t : **TASKS**) {  
 System.***out***.println(**"Setting max cost "** + t.getDuration());  
 t.setLatestStart(**cpLength**);  
 }  
 }  
  
 */\*\*  
 \* Function to get the critical path  
 \** ***@return*** *- Critical path length  
 \*/* **int** getCpLength() {  
 **return cpLength**;  
 }  
}

ClickContextMenu

**import** javax.swing.\*;  
  
*/\*\*  
 \* Class for the context menu object, which opens when the user 'right-clicks'  
 \*/***class** ClickContextMenu **extends** JPopupMenu{  
 **private final** JMenuItem **NEW\_TASK**;  
 **private final** JMenuItem **NEW\_NODE**;  
 */\*\*  
 \* Constructor for ClickContextMenu  
 \* Initialises the options and adds them to the menu  
 \*/* ClickContextMenu(){  
 **NEW\_TASK** = **new** JMenuItem(**"Add Task"**);  
 **NEW\_NODE** = **new** JMenuItem(**"Add/Drag Node"**);  
  
 add(**NEW\_TASK**);  
 add(**NEW\_NODE**);  
 }  
  
 */\*\*  
 \* Get the newNode item  
 \** ***@return*** *- NEW\_NODE item  
 \*/* **public** JMenuItem getNEW\_NODE() {  
 **return NEW\_NODE**;  
 }  
  
 */\*\*  
 \* Get the newTask item  
 \** ***@return*** *- NEW\_TASK item  
 \*/* **public** JMenuItem getNEW\_TASK() {  
 **return NEW\_TASK**;  
 }  
}

Controller

**import** javax.swing.\*;  
**import** java.awt.event.\*;  
*/\*\*  
 \* Controller class used to process mouse and key inputs  
 \*  
 \* Implements action listener to listen to button presses  
 \*/***class** Controller **implements** ActionListener {  
 **private final** GUIMain **VIEW\_MAIN**;  
 **private final** ClickContextMenu **VIEW\_CON\_MENU**;  
 **private boolean drawing**;  
 **private boolean placingNode**;  
 **private boolean selecting**;  
 **private boolean controlHeld**;  
 **private final** PathFunctions **PATH\_FUNCTIONS**;  
 */\*\*  
 \* Constructor for controller  
 \* Creates a PathFunctions object  
 \* Creates a KeyAdapter which listens to 'ctrl' key  
 \* Creates a MouseAdapter to listen to mouse location and click behaviour  
 \* Adds itself as an ActionListener to all buttons  
 \** ***@param guiMain*** *- The main GUI  
 \** ***@param cCMenu*** *- The context menu  
 \*/* Controller(GUIMain guiMain, ClickContextMenu cCMenu){  
 **this**.**VIEW\_MAIN** = guiMain;  
 **this**.**VIEW\_CON\_MENU** = cCMenu;  
 **this**.**PATH\_FUNCTIONS** = **new** PathFunctions();  
 **this**.**drawing** = **false**;  
 **this**.**placingNode** = **true**;  
 **this**.**selecting** = **false**;  
  
 KeyAdapter keyAdapter = **new** KeyAdapter() {  
 @Override  
 **public void** keyPressed(KeyEvent e) {  
 **if** (e.getKeyCode() == KeyEvent.***VK\_CONTROL***) {  
 **controlHeld** = **true**;  
 }  
 **else if** (e.getKeyCode() == KeyEvent.***VK\_DELETE***){  
 System.***out***.println(**"CTRL"**);  
 **VIEW\_MAIN**.getPComponent().deleteSelected();  
 }  
 }  
  
 @Override  
 **public void** keyReleased(KeyEvent e) {  
 **if** (e.getKeyCode() == KeyEvent.***VK\_CONTROL***) {  
 **controlHeld** = **false**;  
 }  
 }  
 };  
 MouseAdapter mainMouseAdapter = **new** MouseAdapter()  
 {  
 @Override  
 **public void** mousePressed(MouseEvent e)  
 {  
 **if** (SwingUtilities.*isLeftMouseButton*(e)) {  
 System.***out***.println(e.getX() + **" , "** + e.getY());  
  
 **if** (!**placingNode**) {  
 **drawing** = **true**;  
 Node n = **VIEW\_MAIN**.startDrawing(**new** Vector(e.getX(), e.getY()));  
 **if** (n == **null**) {  
 **VIEW\_MAIN**.getPComponent().updateSelected(**new** Vector(e.getX(), e.getY()));  
 **drawing** = **false**;  
 }  
 }  
 **else if** (**controlHeld**){  
 System.***out***.println(**"CTRL CLICK"**);  
 **VIEW\_MAIN**.getPComponent().ctrlClick(**new** Vector(e.getX(), e.getY()));  
 }  
 **else if** (e.getClickCount() == 2){  
 **VIEW\_MAIN**.doubleClicked();  
 }  
 **else if** (**placingNode**) {  
 **VIEW\_MAIN**.getPComponent().nodeModeClick(**new** Vector(e.getX(), e.getY()));  
 }  
  
 }  
 **else**{  
 **VIEW\_MAIN**.getPComponent().selectNull();  
 }  
 }  
 @Override  
 **public void** mouseDragged(MouseEvent e) {  
 **if**(SwingUtilities.*isLeftMouseButton*(e)) {  
 **VIEW\_MAIN**.getPComponent().dragSelected(**new** Vector(e.getX(), e.getY()));  
 }  
  
 }  
 @Override  
 **public void** mouseReleased(MouseEvent e) {  
 **if** (**selecting**){  
 **selecting** = **false**;  
 }  
 **else if** (**drawing**) {  
 Task t = **VIEW\_MAIN**.getPComponent().finishDrawing(**new** Vector(e.getX(), e.getY()));  
 **PATH\_FUNCTIONS**.addTask(t);  
 **drawing** = **false**;  
 }  
  
 }  
  
 };  
  
 **VIEW\_MAIN**.getPComponent().addMouseListener(mainMouseAdapter);  
 **VIEW\_MAIN**.getPComponent().addMouseMotionListener(mainMouseAdapter);  
 **VIEW\_MAIN**.getCHECK\_BUTTON().addActionListener(**this**);  
 **VIEW\_MAIN**.getCHECK\_VALID\_BUTTON().addActionListener(**this**);  
 **VIEW\_MAIN**.getCLEAR\_BUTTON().addActionListener(**this**);  
 **VIEW\_MAIN**.getPComponent().setComponentPopupMenu(cCMenu);  
 **VIEW\_MAIN**.getPComponent().addKeyListener(keyAdapter);  
 **VIEW\_CON\_MENU**.getNEW\_TASK().addActionListener(**this**);  
 **VIEW\_CON\_MENU**.getNEW\_NODE().addActionListener(**this**);  
 }  
  
 */\*\*  
 \* Processes button clicks for every button and every phase of each button  
 \** ***@param ae*** *- The action to be processed  
 \*/* **public void** actionPerformed(ActionEvent ae)  
 {  
 **if** (ae.getSource() == **VIEW\_CON\_MENU**.getNEW\_TASK()){  
 **placingNode** = **false**;  
 System.***out***.println(**"Task Mode"**);  
 }  
 **else if** (ae.getSource() == **VIEW\_CON\_MENU**.getNEW\_NODE()){  
 **placingNode** = **true**;  
 System.***out***.println(**"Node Mode"**);  
 }  
  
  
 **else if** (ae.getSource() == **VIEW\_MAIN**.getCHECK\_VALID\_BUTTON()){  
 **switch** (**VIEW\_MAIN**.getCHECK\_VALID\_BUTTON().getText()) {  
 **case "Check Validity Of Network"**:  
 **try** {  
 **PATH\_FUNCTIONS**.criticalPath();  
 **VIEW\_MAIN**.setCRIT\_PATH\_LENGTH(**"Network is valid"**);  
 } **catch** (RuntimeException cyclicDependency) {  
 System.***out***.println(**"Runtime"**);  
 **VIEW\_MAIN**.setCRIT\_PATH\_LENGTH(**"Network is invalid"**);  
 }  
 **break**;  
  
 **case "Show Answers"**:  
 **VIEW\_MAIN**.getPComponent().getD\_LIST().displayAnswers(**true**);  
 **VIEW\_MAIN**.getCHECK\_VALID\_BUTTON().setText(**"Stop Showing Answers"**);  
 **VIEW\_MAIN**.getPComponent().repaint();  
 **break**;  
  
 **default**:  
 **VIEW\_MAIN**.getPComponent().getD\_LIST().displayAnswers(**false**);  
 **VIEW\_MAIN**.getCHECK\_VALID\_BUTTON().setText(**"Show Answers"**);  
 **int** nodeMarks = **VIEW\_MAIN**.getPComponent().getN\_LIST().checkAnswers();  
 **VIEW\_MAIN**.getCHECK\_BUTTON().setText(**"Stop Checking Answers"**);  
 **VIEW\_MAIN**.getCHECK\_VALID\_BUTTON().setText(**"Show Answers"**);  
 **int** taskMarks = **VIEW\_MAIN**.getPComponent().getD\_LIST().checkTaskMarks();  
 **int**[] markArray = **new int**[2];  
 markArray[0] = nodeMarks;  
 markArray[1] = taskMarks;  
 **VIEW\_MAIN**.setMarkAmount(markArray);  
 **VIEW\_MAIN**.getPComponent().repaint();  
 **break**;  
 }  
 }  
  
 **else if** (ae.getSource()==**VIEW\_MAIN**.getCHECK\_BUTTON()){  
 **if** (**VIEW\_MAIN**.getCHECK\_BUTTON().getText().equals(**"Check Answers"**)) {  
 **try** {  
 System.***out***.println(**"Starting algorithm"**);  
 **VIEW\_MAIN**.getPComponent().getN\_LIST().setCheckAnswersFalse();  
 **PATH\_FUNCTIONS**.criticalPath();  
 **VIEW\_MAIN**.setCRIT\_PATH\_LENGTH(**"Critical path length: "** + **PATH\_FUNCTIONS**.getCpLength());  
 **int** nodeMarks = **VIEW\_MAIN**.getPComponent().getN\_LIST().checkAnswers();  
 **VIEW\_MAIN**.getCHECK\_BUTTON().setText(**"Stop Checking Answers"**);  
 **VIEW\_MAIN**.getCHECK\_VALID\_BUTTON().setText(**"Show Answers"**);  
 **int** taskMarks = **VIEW\_MAIN**.getPComponent().getD\_LIST().checkTaskMarks();  
 **int** [] markArray = **new int**[2];  
 markArray[0] = nodeMarks;  
 markArray[1] = taskMarks;  
 **VIEW\_MAIN**.setMarkAmount(markArray);  
 **VIEW\_MAIN**.getPComponent().repaint();  
 }  
 **catch** (RuntimeException re){  
 System.***out***.println(**"Runtime"**);  
 **VIEW\_MAIN**.setCRIT\_PATH\_LENGTH(**"Cyclic dependency in Network"**);  
 }  
 }  
 **else**{  
 **VIEW\_MAIN**.getCHECK\_BUTTON().setText(**"Check Answers"**);  
 **VIEW\_MAIN**.getCHECK\_VALID\_BUTTON().setText(**"Check Validity Of Network"**);  
 **VIEW\_MAIN**.clearMarkLabel();  
 **VIEW\_MAIN**.getPComponent().getN\_LIST().setCheckAnswersFalse();  
 **VIEW\_MAIN**.getPComponent().getD\_LIST().displayAnswers(**false**);  
 **VIEW\_MAIN**.setCRIT\_PATH\_LENGTH(**""**);  
 **VIEW\_MAIN**.getPComponent().repaint();  
 }  
 }  
 **else if** (ae.getSource() == **VIEW\_MAIN**.getCLEAR\_BUTTON()){  
 **PATH\_FUNCTIONS**.deleteAllTasks();  
 **VIEW\_MAIN**.getPComponent().clearComponent();  
 System.***out***.println(**"Clear"**);  
 }  
 **VIEW\_MAIN**.getPComponent().requestFocusInWindow();  
 }  
}

# 

# 4 Testing

## 4.2 Alpha testing

Test plan

|  |  |  |
| --- | --- | --- |
| **Test Number** | **Description** | **Purpose of test, and objectives being tested** |
| 1 | Test that the basic interactive aspects of the GUI work, e.g. putting down Tasks and nodes, and linking tasks with nodes | To ensure that the controller interacts with the GUI and its methods correctly (CO1, CO2, CO4, CO13, CO15, CO17, EO4) |
| 2 | Integration test of the Drawable, testing that Drawable objects all interact with each other correctly | To ensure that creating networks, and interacting with them, is a smooth experience (CO3, CO12) |
| 3 | Unit test, testing that the Nodes and Tasks handle inputs correctly, testing valid, invalid, and boundary inputs | To ensure that tasks can become dummies, marked as critical, and incorrect inputs are dealt with correctly (CO5, CO10, CO11, CO13) |
| 4 | Integration test, testing that the critical path algorithm works for all networks, and inputted answers are marked correctly | To ensure that the users created paths can be marked reliably, no matter what they input (CO6, CO7, CO8, CO9, CO10, CO14, EO2) |
| 5 | Integration test, testing to ensure tasks and nodes can be deleted, and that networks which have had items deleted still work with the algorithm | To ensure that the deleting and clearing functionality works as intended. Also, tests to ensure that networks with multiple critical paths are treated correctly (CO18, CO10, EO7) |
| 6 | Integration test, checking that multiple networks can be marked at the same time | To ensure that the product is flexible to the point of allowing multiple networks |

All errors found in tests will have a detailed report below the tests

Test 1:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Is action valid** | **Expected Outcome** | **Was test successful** | **Reference Screen-shot(s)** |
| Run project | Yes | Main GUI opens in windowed size. Three buttons at the bottom centre of the window “Clear Answers” “Check Validity Of Network” “Check Answers” | Yes | T1a |
| Click on main area | Yes | Node placed, with its centre at mouse location, and its index 0. Node is selected (edge of node is red). Two boxes appear above the node, each with a red zero in them | Yes | T1b |
| Right click on mouse | Yes | Context Menu opens with mouse at top left corner of menu, and gives options. Node is unselected, and its edges, and text, turn black | Yes | T1c |
| Click on add Task | Yes | Context menu closes, program goes into ‘Task mode’ | Yes | T1d |
| Hold left click on node and drag mouse outwards | Yes | Directed line appears, starting at node centre, line end moves with mouse cursor, Line start is behind the node | No, line start was in front of the node (Error now resolved) | T1e |
| Release left click at some distance from starting node | Yes | Node appears at end of Task, duration of ‘1’ appears above Task centre, the created Node has an index of ‘1’, Created task is selected red, task is behind both nodes | Yes | T1f |
| Draw a task from the first node to a location very close to the second node | No | Upon clicking on the first node, output says valid start, upon releasing click next to the node, output says invalid end, no task created | Yes | T2g |
| Re-size window | Yes | Buttons move with window, main component enlarges with nodes and task staying in the same position | Yes | T1h |
| Set window to full screen | Yes | Window becomes full screen (with borders), buttons are now at bottom centre of window | Yes | T1i |
| Close the application | Yes | Window closes. Program halts | Yes |  |

Test 2:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Is action valid** | **Expected Outcome** | **Was test successful** | **Reference Screen-shot(s)** |
| Using the context menu, enter Task Mode | Yes | Task mode entered, as shown by printed text | Yes | T2a |
| Click on empty space, attempting to put a task down | No | No task appears, printed text states that action is invalid | Yes | T2b |
| Using the context menu, enter Node Mode | Yes | Node mode entered, as shown by printed text | Yes | T2c |
| Place a node | Yes | Once first node is placed it is selected and has an index of 0 | Yes | T2d |
| Click close to, but not touching the node, attempting to place another node intersecting the first | No | A node is not placed, the first node is unselected (Now has a black border) | Yes | T2e |
| Click at a greater distance to the first node, placing a second | Yes | The second node has an index of 1, and is selected (red border) | Yes | T2f |
| Click on the first node, selecting it | Yes | The first node becomes selected, and the second is unselected | Yes | T2g |
| Hold the mouse down on the second node, and move the mouse | Yes | Node is dragged with the mouse, its centre staying on the mouse tip | Yes | T2h |
| Drag the second node so that its circle touches the first node’s circle | Yes | The nodes are allowed to touch, and don’t flicker | Yes | T2i |
| Attempt to drag the second Node through the first | No | Instead of intersecting, the first node moves around the second, keeping itself touching the second at a point which extends the vector from the centre of the second node to the mouse tip | Yes | T2j |
| Move the first node to the edge of the component at all sides and corners, attempt to drag it out of the window | No | Node stays at the edge of the component | Yes | T2k, T2l |
| Keeping the first node at the edge of the component, attempt to drag the second node through the first | No | The second node, instead of going around the first and out of the component, stays at the edge of the node until the point where it appears on the other side of the first node | No (Error now resolved) | T2m |
| Extend the window further right and attempt to drag the first node through the right hand edge of the component |  | Node stays at the edge of the component | Yes | T2n |
| Click at distance from both nodes, attempting to create a third node | Yes | Third node created with index 2, the selected node changes to the third | Yes | T2o |
| Using the context menu, enter task mode | Yes | Task mode entered, as shown by printed text | Yes | T2p |
| Attempt to create a task which starts and finishes at the first node (by dragging and releasing mouse over the node) | No | Output of “Task cannot start and end at the same node”. No task is created | Yes | T2q |
| Click and drag from the first node to the third node, then from the second node to the third node | Yes | A task is drawn from the first to third node, and is selected (red outline). The third node is unselected . Another task is drawn from the second to third node, and is selected, the first task is unselected (black outline) | Yes | T2r, t2s |
| Re-enter node mode, hold click on the third node, and move the mouse | Yes | The third node is selected and the second task is unselected, as the third node is dragged, the two connected tasks follow it | Yes | T2t |
| Drag the second and third nodes so that they touch, then attempt to drag the first node through where they touch | No | Third node stays touching the first and second, until the mouse move past the centre of the intersection, where the third node flips side | Yes | T2u, T2v |
| Drag the first node across the window | Yes | Task connected to the first node changes its starting position with the node | Yes | T2w |
| Create a task from the third node to the first node | No | Output of "Task already found between these nodes". No task created | Yes | T2x |
| Press the clear all button | Yes | All tasks and node disappear instantly | Yes | T2y |

Test 3:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Is action valid** | **Expected Outcome** | **Was test successful** | **Reference Screen-shot(s)** |
| Place a node, and double click on the upper left box | Yes | After the second click, a window will open, prompting the user to input the node’s early event time | Yes | T3a |
| Enter the number 7 | Yes | Once entered, and confirmed, the 0 in the upper left box will become a 7 | Yes | T3b, T3c |
| Repeat the action, but enter the number -1 | No | The 7 will stay in the box, and there will be a statement saying invalid input | Yes | T3d |
| Repeat the action, but enter 4.3 | No | The 7 will stay in the box, and there will be a statement saying invalid input | Yes | T3e |
| Repeat the action, but enter ‘a2s’ | No | The 7 will stay in the box, and there will be a statement saying invalid input | Yes | T3f |
| Repeat the action, but enter 0 | Yes (Boundary) | The 7 will be replaced by ‘0’ upon confirmation | Yes | T3g |
| Double click the upper right box | Yes | After the second click, a window will open, prompting the user to input the node’s late event time | Yes | T3h |
| Repeat the earlier inputs, this time in the late event time prompt window | 7 – yes  -1 – no  4.3 – no  a2s – no  0 – yes | If the input is valid, the number will change, if not, it will not. A message will appear if invalid  When the duration is 0, the line will become dashed, as it is now a dummy | Yes | T3i, T3j, T3k, T3l, T3m |
| Enter task mode, and draw a task from the first node to empty space, creating a second node. Enter Node Mode, and  hold ‘ctrl’ and click on the line of the task | Yes | Once ‘ctrl-clicked’, the task line will turn blue, as it is marked as critical | Yes | T3n |
| Double click on the task line | Yes | After the first click, the line will go red. After the second click, a window will open, prompting the user to input a duration for the task. | Yes | T3o |
| Try all the inputs listed earlier (without 0), this time on the task | 7 – yes  -1 – no  4.3 – no  a2s – no | If the input is valid, the number above the task will change, if not, it will not. A message will appear if invalid | Yes | T3p, T3q, T3r, T3s |
| Ctrl click the task again | Yes | The task will return to having a black line, as it is no longer marked as critical | Yes | T3t |
| Double click on the task, and enter 0 into the option pane | Yes (Boundary) | Task should now have a duration of 0, its line will be dashed, and its duration hidden. It is now a dummy | Yes | T3u |
| Double click on the dummy task and enter 43 | Yes | Task should return to having a solid line, and its duration, 43 will now be shown | Yes | T3v |
| Insert another task between the second node and a third node (With the two tasks being parallel to each other), then click on a node to unselect it. Finally click on the new task | Yes | When clicking on the new task, the old task is selected | No (Error now resolved) | T3w, T3x |
| Double click on the first task to open the option pane, and click cancel | Yes | Option pane closes, task duration remains the same | Yes | - |
| Ctrl click the first task, and set its duration to 0, then unselect the task by clicking on a node | Yes | Upon becoming a dummy, the task will revert back to being black (Dummy tasks cannot be critical) | Yes | T35, T3z |

Test 4:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Is action valid** | **Expected Outcome** | **Was test successful** | **Reference Screen-shot(s)** |
| Create the diagram shown in Test diagram 1 | - | - | - | T4a |
| Click the button labelled “Check Validity of network” | Yes | Text appears on the GUI stating “Network is valid” | Yes | T4b |
| Click the button labelled Check Answers | Yes | Network should have the same colours as Test Diagram 2  Critical path length of 27 is stated to the right of the check answer button. Checking answers button now reads “Stop checking answers”. At the top of the component, in the middle, a label reads “2 / 4 for Nodes, 1/2 for Tasks, 3/6 total” | No, most were marked correctly, however, there were some incorrect, the  mark amount was incorrect | T4c |
| Click the “Stop checking answers” button | Yes | Network now looks like network 1 again. Critical path length is hidden an button returns to saying “Check answers” | Yes | T4d |
| Change the late event time of node 5 to 9, and the early event time of node 6 to 14,  Then click the “Check answers button” | Yes | Late event time of node 5 is marked red, whilst early event time of node 6 is green | Marking was correct on the changed nodes so yes | T4e |
| Click the show answers button | Yes | Network now has all correct answers written in blue writing in the box, button now reads “Stop Showing Answers”, critical path marked blue | No, node 6 had incorrect late event time, as did node 0 and node 3. Critical path incorrectly marked | T4f |
| Click the stop showing answers button | Yes | Network returns to having the same colours and fields as diagram 3 | Yes | T4g |
| Click the “Stop checking answers” button | Yes | Network returns to having ‘normal’ colours | Yes | - |
| Draw a task from node 7 to node 3 | - | - | - | T4h |
| Click the button labelled “Check Validity of network” | Yes | Text appears on the GUI stating “Cyclic dependency in network” | Yes | T4i |
| Click the check answer button | No | Text appears on the GUI stating “Cyclic dependency in network” | Yes | T4j |

## Test 5: (Performed after errors found in tests 1, 2, 3, and 4 were fixed)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Is action valid** | **Expected Outcome** | **Was test successful** | **Reference Screen-shot(s)** |
| Create the diagram shown in test diagram 3, do not fill in nodes | - | - | - | - |
| Click the check answer button and then the show answer button | Yes | Boxes are filled in as shown in test diagram 4. Critical tasks in diagram 4 are marked as blue. Crit path length is 38 | Yes | T5a |
| Click stop checking answers Enter Node Mode, and click to select node 4 | Yes | Node 4 selected | Yes | T5b |
| Press the delete key | Yes | Node 4 is deleted, along with the tasks directly connected to it | Yes | T5c |
| Select the task joining nodes 6 and 8, and press the delete key | Yes | The task is removed | Yes | T5d |
| Click the button labelled “Check Validity of network” | Yes | Text appears on the GUI stating “Network is valid” | Yes | T5e |
| Click the check answer button and then the show answers button | Yes | Network now has all correct answers written in blue writing in the boxes. Critical path adjusted so that the task joining 14 and 12 is now critical. Crit path length is now 36 | Yes | T5f |
| Click the “Stop checking answers” button | Yes | Network returns to having ‘normal’ colours | Yes | T5g |
| Press the clear all button | Yes | All tasks and nodes are deleted | Yes | T5h |
| Create a network of a triangle of tasks, with durations 1, 3, and 2. Click check answers, and then click display answers | Yes | Note, node indexes should start counting from 0 after clearing.  Numbers shown and tasks which are critical are correct | Yes | T5i |

Test 6:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Action** | **Is action valid** | **Expected Outcome** | **Was test successful** | **Reference Screen-shot(s)** |
| Create the two networks shown in test diagram 4 | - | - | - | - |
| Click the button labelled “Check Validity of network” | Yes | Text appears on the GUI stating “Network is valid” | Yes | T6a |
| Click the check answer button and then the show answers button | Yes | Network now has all correct answers written in blue writing in the box, shown in test diagram 5. Button now reads “Stop Showing Answers”. Critical path length is 7. (Note that networks should be marked as if they were both part of one larger network) | Yes | T6b |
| Add a task from node 8 to node 5. Then click the button labelled “Check Validity of network” | Yes | Text saying “Cyclic dependency in Network” | Yes | T6c |

## 4.2.2 Errors discovered:

|  |  |  |
| --- | --- | --- |
| **Error** | **Cause of error and fix** | **Screenshots** |
| Whilst drawing a task from a node, the task is drawn over the node, the nodes should always appear on top of tasks | In the drawObjects procedure, the procedure to draw every existing task and node is ran before the code which draws the task currently being made.  Due to the way java works, this means that the task overlaps the nodes, as shown in E1a.  I have now amended this by changing the order of how things are drawn, as displayed in the amended code section | T1e - Error  T1e After fix - Fixed |
| If you drag a node through another on the edge of the component, the dragged node can go through the edges of the components | The error is being cause by the moveToNode procedure. This procedure is used to move a node which would usually intersect another node around the intersecting node. The issue being that when the procedure found a new centre away from the intersecting node, it was applied to the first node without being checked to see if it would push the node outside the boundaries of the component.  I have amended this by making the program check the potential new centre to see if it would cross the boundaries, if it crosses then the change in centre isn’t applied | T2m - Error  T2m After fix - Fixed |
| When you attempt to click on a task or any part of the component, if this click location intersects a line which extends from either ends of an existing task, the click will register as intersecting this task. This means that tasks are being selected incorrectly | The cause of this error was the perpendicularDistanceToLineSeg function. This uses vector mathematics to find the shortest distance between a point and a line. This is the function used to see whether a click is close enough to a task to be intersecting. The problem was that the function was checking the perpendicular distance to the whole line, and not just the segment occupied by the task.  To fix this, I simply added an if statement to check whether the magnitude of the line from the start of the task to the closest point on the line to the vector is within the range of 0 <= p <= 1  With 1 being the length of the task and p being the multiple. If outside this range, then the point must be too far away from the task to be selected | T3x - Example of error  T3x After fix - Fixed |
| Seemingly random nodes have their late event time marked incorrectly, and the incorrect marking is inconsistent. | Identifying this error was difficult; the only hint was that the incorrectly marked nodes were always nodes with more than one succeeding task. The event times were calculated using the early start and latest start of one of the nodes succeeding tasks. The issue being that multiple tasks can start at one node, and they could both have different latest start times, depending on where they go to and their duration. The correct latest start time being the lowest of all the tasks  I have changed the code now so the early and late event times of a node are calculated at the same time as the early and late start times of the tasks, and if a task has shorter finish time, the nodes event times are overwritten. This not only fixes the issue, but makes the calculations more time effective. | T4h - Error  T4h After fix - fixed |
| Some critical tasks are not being identified as critical, whilst some non-critical tasks are seen as critical | This was also caused by the incorrectly marked late event times, as the tasks are calculated as critical based on the event times of nodes. So if a node had an incorrect late event time it would be in danger of having its connected tasks marked incorrectly | T4h - Error  T4h After fix - fixed |

Altered code to fix errors:

Error 1:

Updated drawObjects procedure:

**private void** drawObjects(Graphics g) {  
 **if** (**drawing**){  
 g.setColor(**new** Color(0,0,0));  
 g.drawLine(**x1**, **y1**, **x2**, **y2**);  
 Graphics2D g2d = (Graphics2D) g;  
 g2d.draw(**new** Vector(**x1**, **y1**).getIntersectingLine(**new** Vector(**x2**, **y2**), 0.7, 20, 30)); *//Draws arrow* g2d.draw(**new** Vector(**x1**, **y1**).getIntersectingLine(**new** Vector(**x2**, **y2**), 0.7, 20, -30));  
 }  
 **D\_LIST**.drawAll(g);  
}

Error 2:

Updated node movement code with boundary checks:

**if** (!(**selectedDrawable** == **null**)) {  
 **if** (**selectedDrawable**.getObjectPriority() == 2) {  
 Node selected = (Node) **selectedDrawable**; *//Safe to cast because selectedDrawable is proven to be a node* Node collidingNode = **N\_LIST**.circleIntersectCheck(v, selected.getINDEX());  
 **if** (collidingNode == **null**) {  
 selected.setCentre(v);  
 }  
 **else** {  
 Vector potentialCentre = selected.moveToNode(v, collidingNode); *//Tries to make new centre which doesn't intersect node* **if** (**N\_LIST**.circleIntersectCheck(potentialCentre, selected.getINDEX()) == **null**) {  
 **if** ((potentialCentre.getIntX() < componentSize.getWidth() - 31) && (potentialCentre.getIntX() > 30)){ *//Boundary checks* **if** ((potentialCentre.getIntY() < componentSize.getWidth() - 21) && (potentialCentre.getIntY() > 50)){  
 selected.setCentre(potentialCentre);  
 }  
 }  
 }  
 }  
 }  
}

Error 3:

Updated perpendicularDistanceToLineSeg function:

**double** perpendicularDistanceToLineSeg(Vector start, Vector end){  
 Vector startToPoint = **this**.subtract(start);  
 Vector line = end.subtract(start);  
 **double** p = -1;  
 **if** (line.getMagnitudeSquared() != 0){  
 p = startToPoint.getDotProduct(line) / line.getMagnitudeSquared();  
 }  
 **if** ((p >= 0) && (p <= 1)) {  
 Vector d = **this**.subtract(start.add(line.multiply(p)));  
 **return** d.getMagnitude();  
 }  
 **return** 4000; *//Return large number if p not in range*}

Error 4 and 5:

Updated checkAnswers loop:

**while** (current != **null**){  
 **if** (current.isConnected()) {  
 **if** (current.getSucceedingTasks().isEmpty()) {  
 Iterator<Task> it = current.getPrecedingTasks().iterator();  
 Task t = it.next();  
 current.setCalculatedEarlyStart(t.getLatestStart() + t.getDuration());  
 current.setCalculatedLatestStart(t.getLatestStart() + t.getDuration());  
 }  
 System.***out***.println(current.getINDEX() + **"Is connected"**);  
 current.setCheckingAns(**true**);  
 marks += current.getMarksForNode();  
 nodeCount++;  
 }  
 current = current.getNextNode();  
}

Updated setEarly function:

**private void** setEarly(Task t1) {  
 **int** completionTime = t1.getEarlyFinish();  
 **for** (Task t2 : t1.getTASK\_VISUALISED().getEND\_NODE().getSucceedingTasks()) {  
 **if** (completionTime >= t2.getEarlyStart()) {  
 System.***out***.println(**"Setting early start and finish "** + t1.getDuration());  
 t2.setEarlyStart(completionTime);  
 t2.setEarlyFinish(completionTime + t2.getDuration());  
 setEarly(t2);  
 }  
  
 **if** (t2.getTASK\_VISUALISED().getSTART\_NODE().getCalculatedLatestStart() == 0) {  
 t2.getTASK\_VISUALISED().getSTART\_NODE().setCalculatedLatestStart(t2.getLatestStart());  
 }  
  
 **else if** (t2.getTASK\_VISUALISED().getSTART\_NODE().getCalculatedLatestStart() > t2.getLatestStart()) {  
 t2.getTASK\_VISUALISED().getSTART\_NODE().setCalculatedLatestStart(t2.getLatestStart());  
 }  
  
 **if** (t2.getTASK\_VISUALISED().getSTART\_NODE().getCalculatedEarlyStart() < t2.getEarlyStart()) {  
 t2.getTASK\_VISUALISED().getSTART\_NODE().setCalculatedEarlyStart(t2.getEarlyStart());  
 }  
 }  
}

# 5 Evaluation

## 5.1 Objectives

Note, that objectives concerning the opinion of the user, are evaluated in the third party feedback section

|  |  |
| --- | --- |
| **Core Objective** | **How the objective has been met, and tests proving this** |
| The user must be able to create a task, specifying the length of the task as well as its location | Using the right click menu to enter task mode, a user can place a task of length 1 starting at any node, this task can then have its length changed at any point by double clicking it. Test 1, Test 3 |
| The user must be able to create a node, which will be given a unique identifier; the first node will have an identifier of 1, the second 2, and so on…. The user must be able to specify a nodes location, as long as it doesn’t intersect another node | Using the right click menu to enter node mode, a user can click anywhere on the component to place a node, the indexes work as stated in the objective. If the user attempts to place a node in a position where it will intersect another node, the node will not be placed. Users can also place a node by drawing a task from a pre-existing node to empty space. If there is room at the end of the task, a node is created. Test 1 |
| Placed nodes must be able to be dragged across the component, attached tasks must move with the nodes | Whilst in node mode, by holding left click on a node and moving the mouse, the node will be dragged across the component, its centre staying on the mouse. All tasks directly attached will move so that their start/end stays on the nodes centre. Nodes cannot be dragged into other nodes, and if possible, will move around other nodes. Nodes cannot be dragged out of the component Test 2 |
| The diagram of the tasks must be in a format similar to the taught format for the A level specification, which wouldn’t alienate the user | The nodes follow the form of most of the questions in the A-level exams. Dummy tasks look as they do in exams, as do tasks. All tests show this |
| The user must be able to input their calculated values for the early and late event times for each node | By double clicking in each box of a node, you can change the value for that box. With the left box being the early event time and the right being the late event time. Invalid values are dealt with correctly. Test 3 |
| The program must have a method of calculating the early and late event times for every node in any given network | The path functions class, in conjunction with the Task objects, can do a sweep of any network created, and if the network is valid, will calculate event times for every node which is part of the network. Test 4 , test 5, test 6 |
| The program must have a method of calculating which tasks are critical in any given network | Using the calculated event times, the program can calculate the float of tasks, and if the task has a float of zero, and isn’t a dummy, its identified correctly as critical. Test 4, Test 5, Test 6 |
| The program must have a method of calculating the critical path length of any given network | The path functions class finds a ‘mas cost’ of any network, which is equal to the critical path length. The critical path length is shown at the bottom of the window when the user is showing answers. Note, with multiple networks, the shown critical path length is the shortest time to complete every task on every network. Test 4, test 5, test 6 |
| The program must have a method of showing the user the correct early starts and the correct late finishes for all connected nodes, making clear whether the user inputted correctly | By clicking the Check Answers button, every node connected to a network is marked. If the users inputted value is equal to the actual value, the box the value is in is shown as green. Otherwise, the box is red. Whilst on the check answers mode, a show answers button appears, this button shows the user a correctly labelled version of the networks they created Test 4, test 5, test 6 |
| The program must have a method of allowing the user to identify tasks as critical, and showing whether tasks are identified correctly | Tasks can be identified as critical by holding Ctrl and clicking on the task whilst in node mode. Repeating this action, un identifies it. Identified tasks are coloured blue. Upon clicking the check answers button, correctly identified tasks stay whichever colour they were, incorrectly identified tasks turn red Test 3, test 4, test 5, test 6 |
| The user must be able to edit the length of tasks | By double clicking a task, a user can change its duration. Invalid values are handled correctly. Test 3 |
| When creating tasks, a task must only be allowed if it starts and ends at a node | Tasks cannot start in empty space, if a user tries to make a task, which ends in empty space, the program will attempt to place a node at the task’s end. If a node will not fit at the task’s end, the task is not created. Test 1 |
| The user must be able to create a dummy task, which will be drawn as a dashed line, making it distinguishable from regular tasks | Upon setting a tasks duration to 0, it becomes a dummy task. It now has a dashed line and it does not state a duration, dummy tasks cannot be marked as critical. Test 3 |
| The program must have an option to verify whether the diagram which the user has drawn is valid, this would verify that all the tasks are linked to nodes, that there are no tasks starting and finishing at the same nodes as another task, and that the path contains no cycles. Another option must show the correct answers for only what the user has inputted | There is a button to check whether networks are valid without displaying answers. As proven in test 2, you cannot create a task, which starts and ends at the same node, or create a task which is between the same two nodes as another task. The check if valid button ensures that there are no cycles in the network. Test 2, test 4, test 5 |
| The program must scale to the size of the user’s monitor automatically | The window initially boots up at a set size, however, the user can enlarge the window to full screen, scaling it to the size of the users monitor. When enlarged, the scale of tasks, nodes, and buttons, is not affected. The buttons keep their position in the centre of the window. Test 1 |
| The user must be able to delete nodes and tasks, deleting a node must delete directly connected tasks | The user can delete a Drawable by selecting a node or a task, and pressing the delete key. Directly connected tasks are deleted with nodes. Further, a user can click a button to clear every node and task, which resets the node indexing. Deleted tasks are not considered by the algorithms to calculate event times and path length. Test 5, test 6 |

Extension objectives:

|  |  |
| --- | --- |
| **Extension Objective** | **How the objective has been met, and tests proving this** |
| The program could be altered to add a new option which allows the user to switch to a Gantt chart view, which shows the added activities in the form of a scheduling chart  The added Gantt chart mode will allow the user to move activities within their floats  The tasks on the Gantt chart should be moveable by the user onto different rows, allowing the user to practice rescheduling activities | Gantt chart functionality was not added in the final program, due to a lack of time. If I had more time, I would have included more scheduling |
| For the program to give the user marks out of a total when they check their answers, giving the user more incentive to learn and improve. The amount of marks awarded should be accurate in respect to how many marks are awarded and what merits each mark (4 marks for correct event times, 2 marks for stating critical activities) | Marks are calculated whenever the user checks their answers. The marks do follow the guide written in the objective. The marks are calculated by a ratio of how many inputs are correct: how many possible inputs there are. Note that dummy tasks do not affect this ratio when it comes to calculating the mark out of 2 for critical activities, as they are not real tasks. Test 4, test 5, test 6 |
| For the user to have the option to change the theme of the program, this would adjust background colour as well as the colour of the tasks | Theming was not added to the solution, due to a lack of time, since it was a low priority to add. However it is a good idea for maintenance as it would be quite simple to add |
| The window of the program may be altered to become re-scalable; re-scaling must keep the UI intact and all options viewable | The window is fully re scalable, the UI is kept intact whenever the window is rescaled. Test 1, test 2 |
| The program could have a few pre-made paths to show the user examples of what they could make | There are no pre made paths in the current version. Again, this could be added in future to help demonstrate the project |
| The program could be adjusted to pan the task window, and zoom in and out of it | The camera cannot be controlled at the moment, as it would have taken a long time to implement a movable camera, and edit all the collisions, since they are based on distances |
| The program could be lenient and allow multiple paths to be created and marked at one instance | Multiple networks can be created and marked simultaneously. As stated earlier, the critical path length is the length of the longest network, as the program treats all the networks as one larger network, connected by the end node. Test 6 |

## 5.1 Third party feedback and analysis

I asked the following questions to the user in the form of an online survey:

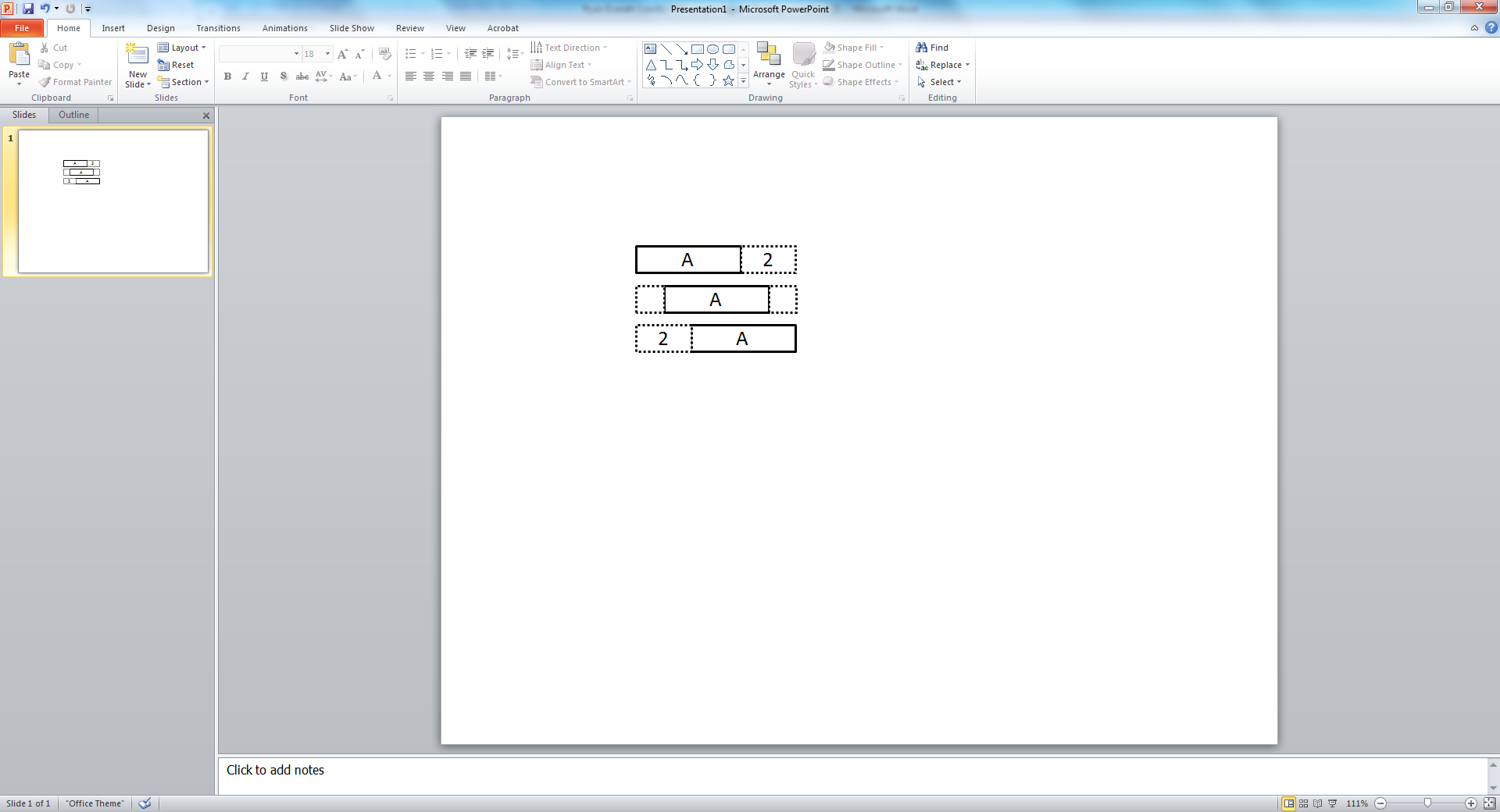
|  |  |
| --- | --- |
| **Question** | **Response** |
| In your opinion, are the controls for creating diagrams are intuitive, and easy to learn? | “Yes, I feel the controls are relatively easy to learn and allow me to do what I want with the program. My only suggestion is to add a tool that only selects as at times I find myself placing nodes when I am trying to edit the weight of an edge.” |
| In your opinion, is the look of the window professional? | “The look of the window is minimal and clean giving it a professional feel. All the information is shown clearly and makes it easy for me to understand.” |
| Has the program marked your answers for inputs correctly? | “Yes, my answers have been marked correctly including the critical path. These are displayed clearly in red or green.” |
| Do created diagrams look like they do in the mock exam papers? | “Yes they look exactly like ones I have seen before in mock papers.” |
| In your opinion, have all the core objectives been met by the solution? Explain | “Yes, a large majority, if not all, core objectives have been met clearly and been well executed within the program.” |
| Are there any extra features you would like to see added, which you feel should have been in the solution? | “It would be nice to see an undo/redo feature as sometimes I would accidentally delete or change something I didn't mean to.” |

Overall, the feedback received from the user was good. I noticed that they would like to have a tool, which only selects; this could be easily implemented, by adding a tertiary mode of use to the controller, which would only run the selection procedure in the PathComponent class. The other improvement, which they would like to see, is undo-redo functionality. To implement this, I would create a stack object of all the operations the user has done, and each operation can be undone/redone in order, as a stack is a ‘last in first out’ structure.

## 5.2 Further future improvements

As stated earlier, in future, I could implement a Gantt chart mode. To create this mode, I would first need to add a second component to the Main GUI, which would replace the path component when toggled. This second component would need to be able to access a list of all the tasks currently in the window, since they would need to be added to the chart. The actual chart would simply be made of line segments and rectangles. The line segments will lie behind the rectangles, and represent the days of the project. (With days being the start date, given that a task with duration of 2 takes 2 days). The rectangles would represent the tasks.

To distinguish between the tasks on the chart, every non-dummy activity must have a unique letter assigned to it. These letters must be assigned on creation of the task, as it needs to be seen by the user so they know which task is which.

The height of every rectangle would be the same, but the length would depend on the length of the task. For example, a task with duration of 9 would be 3 \* longer than a task with a duration of 3. The float of each activity (calculated by late event time of end node – early event time of start node – duration), would be represented by a rectangle with dashed line edges, extending from the right hand side of the rectangle for the task. The task would then be draggable, using the same logic as dragging node, so that it can move into the float. In this scenario, the float rectangle stays in position, and there is now float on the left hand side of the task. A task can only be dragged so it starts at integer amount of days. For example, a Task ‘A’, with duration 5, and float of 2 could be in these stages…

Another improvement to consider would be adding pre made networks. A way to implement this would be to introduce saving and loading features. When ‘saved’, the drawable list would be stored using object serialisation. So when a network is loaded, its drawableList is loaded into the project, and a node list would be created from the nodes in the drawable list. The drawableList can then be painted onto the component, and its tasks will be added to the path functions object.

# 6 Appendices

text