|  |  |  |
| --- | --- | --- |
|  | **FACULTY OF COMPUTING, ENGINEERING and SCIENCE** | Final mark awarded:\_\_\_\_\_ |

**Assessment Cover Sheet and Feedback Form 2016/17**

|  |  |  |  |
| --- | --- | --- | --- |
| Module Code:  **CS4S703** | Module Title:  **Advanced Data Structures and Algorithms** | | Module Lecturer:  **Bertie Müller** |
| Assessment Title and Tasks: **Petri Net Algorithms** | | | Assessment No. **1 out of 2** |
| No. of pages submitted in total including this page:  14 | | | Word Count of submission  (if applicable): Completed by student |
| Date Set:  21 November 2016 | | Submission Date:  23 January 2017 | Return Date:  13 February 2017 |

|  |  |
| --- | --- |
| ***Part A: Record of Submission (to be completed by Student)*** | |
| **Extenuating Circumstances**  If there are any exceptional circumstances that may have affected your ability to undertake or submit this assignment, make sure you contact the Advice Centre on your campus prior to your submission deadline. | |
| **Fit to sit policy**:  The University operates a fit to sit policy whereby you, in submitting or presenting yourself for an assessment, are declaring that you are fit to sit the assessment. You cannot subsequently claim that your performance in this assessment was affected by extenuating factors. | |
| **Plagiarism and Unfair Practice Declaration:**  By submitting this assessment, you declare that it is your own work and that the sources of information and material you have used (including the internet) have been fully identified and properly acknowledged as required[[1]](#footnote-1). Additionally, the work presented has not been submitted for any other assessment. You also understand that the Faculty reserves the right to investigate allegations of plagiarism or unfair practice which, if proven, could result in a fail in this assessment and may affect your progress. | |
| **Intellectual Property and Retention of Student Work:**  You understand that the University will retain a copy of any assessments submitted electronically for evidence and quality assurance purposes; requests for the removal of assessments will only be considered if the work contains information that is either politically and/or commercially sensitive (as determined by the University) and where requests are made by the relevant module leader or dissertation supervisor. | |
| **Details of Submission:**  Note that all work handed in after the submission date and within 5 working days will be capped at 40%[[2]](#footnote-2). No marks will be awarded if the assessment is submitted after the late submission date unless extenuating circumstances are applied for and accepted (Advice Centre to be consulted). | |
| You are required to acknowledge that you have read the above statements by writing your student number(s) in the box: | Student Number(s):  **13010107** |

**IT IS YOUR RESPONSIBILITY TO KEEP A RECORD OF ALL WORK SUBMITTED**

|  |
| --- |
| **Part B: Marking and Assessment**  **(to be completed by Module Lecturer)** |
| This assignment will be marked out of 100%  This assignment contributes to 50% of the total module marks.  This assignment is bonded. |
| **Assessment Task:**  See attached at the end of Part B. |
| **Learning Outcomes to be assessed** (as specified in the validated module descriptor <http://icis.glam.ac.uk>):   * demonstrate a critical understanding of the concepts associated with applying and implementing a variety of data structure algorithms * to create data structures and appraise their role in producing software solutions to non-trivial programming problems |
| **Grading Criteria:**     |  |  |  | | --- | --- | --- | | **Marking Scheme** | **Marks Available** | **Marks Awarded** | | Petri Net model for the algorithm created in Renew | 30 |  | | Demonstration and critical evaluation of your Petri net model | 20 |  | | Correctness and suitability of the C++ implementation of your Petri net model | 20 |  | | Maintainability of the C++ code | 10 |  | | Demonstration and critical evaluation of your C++ code | 20 |  | | **Total** | **100** |  |  |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **Assessment Criteria** | | | | | | | |  | | **Fail (0-29)** | **Narrow Fail (30-39)** | **3rd Class/Pass (40-49)** | **Lower 2nd Class / Pass (50-59)** | **Upper 2nd Class / Merit (60-69)** | **1st Class / Distinction (70 -100)** | | | Petri Net model for the algorithm created in Renew (30) | | ▫PN model does not capture/solve the problem.  ▫PN net does not run in Renew. | ▫PN model captures/solves the problem only partially.  ▫PN net runs in Renew, but shows significant flaws. | ▫PN model captures/solves the problem in a very basic form and with some flaws.  ▫PN net runs in Renew. | ▫PN model captures/solves the problem making but does not facilitate nested PNs to their full potential.  ▫PN net runs in Renew with good attempts at visualising important aspects. | ▫PN model captures/solves the problem making good use of nested PNs.  ▫PN net runs in Renew with good visualisation of important aspects. | ▫PN model captures/solves the problem making excellent use of nested PNs.  ▫PN net runs in Renew with excellent visualisation of important aspects.. | | | Demonstration and critical evaluation of your Petri net model (20) | | ▫No properties of the net are discussed.  ▫The PN model is completely ad hoc.  ▫Workings of the PN are not explained in sufficient detail.  ▫No evidence of testing. | ▫Some properties of the net are discussed but with significant flaws.  ▫The PN model is somewhat ad hoc.  ▫Explanation of workings of the PN attempted but sometimes incorrect.  ▫Too little evidence of testing. | ▫Most properties of the net are discussed but with some flaws.  ▫The PN model is mostly well designed.  ▫Explanation of workings of the PN attempted and mostly correct.  ▫Some evidence of testing. | ▫Most properties of the net are correctly discussed.  ▫The PN model is mostly well designed, but basic.  ▫Explanation of workings of the PN attempted and correct, but may omit some detail.  ▫Intermediate level of testing. | ▫All important properties of the net are correctly discussed.  ▫The PN model is well designed.  ▫Explanation of workings of the PN are correct and complete.  ▫Good level of testing. | ▫All important and some less important properties of the net are correctly discussed.  ▫The PN model shows excellent design throughout.  ▫Explanation of workings of the PN are correct, complete, and concise.  ▫Excellent level of testing | | | Correctness and suitability of the C++ implementation of your Petri net model (20) | | ▫C++ code is not an adequate representation of the PN.  ▫C++ code does not solve the problem. | ▫C++ code attempts representation of the PN but has significant flaws.  ▫C++ code has serious deficiencies in providing a solution to the problem. | ▫C++ code attempts representation of the PN but has minor flaws.  ▫C++ code has some deficiencies in providing a solution to the problem. | ▫C++ code attempts representation of the PN.  ▫C++ code has at most minor deficiencies in providing a solution to the problem. | ▫C++ code is a good representation of the PN.  ▫C++ code solves the problem. | ▫C++ code is an excellent representation of the PN.  ▫C++ code solves the problem in an excellent way. | | | Maintainability of the C++ code (10) | | ▫C++ code does not comply with the framework from the lectures.  ▫C++ code does not follow OOP guidelines.  ▫C++ code does not follow naming conventions.  ▫No comments. | ▫C++ code does not comply with the framework from the lectures in some parts.  ▫C++ code does not always follow OOP guidelines.  ▫C++ code does not always follow naming conventions.  ▫Insufficient comments. | ▫C++ code mostly complies with the framework from the lectures.  ▫C++ code mostly follows OOP guidelines.  ▫C++ code mostly follows naming conventions.  ▫Mostly adequate comments. | ▫C++ code complies with the framework from the lectures.  ▫C++ code follows OOP guidelines.  ▫C++ code follows naming conventions.  ▫Adequate comments. | ▫C++ code complies with the framework from the lectures in all parts.  ▫C++ code follows OOP guidelines to a good degree.  ▫C++ code follows naming conventions to a good degree.  ▫Good commenting. | ▫C++ code complies with the framework from the lectures in an excellent way.  ▫C++ code follows OOP guidelines excellently.  ▫C++ code follows naming conventions to an excellent degree.  ▫Excellent commenting. | | | Demonstration and critical evaluation of your C++ code (20) | | ▫Program does not run or crashes.  ▫The implementation is not explained in sufficient detail in either the report or the code demo. | ▫Program does not run correctly or crashes sometimes.  ▫The implementation is not well explained in the report or the code demo. | ▫Program runs but might crash sometimes.  ▫The implementation is explained with a basic level of detail in the report or the code demo. | ▫Program runs correctly.  ▫The implementation is explained with a basic level of detail in the report and the code demo. | ▫Program runs correctly and with good UI.  ▫The implementation is explained with a good level of detail in the report and the code demo. | ▫Program runs correctly and with excellent UI.  ▫The implementation is explained with an excellent level of detail in the report and the code demo. | | |

|  |  |  |
| --- | --- | --- |
| **Feedback/feed-forward** (linked to assessment criteria):   * Areas where you have done well: * Feedback from this assessment to help you to improve future assessments: * Other comments | | |
| **Mark:** | **Marker’s Signature:** | **Date:** |
| **Work on this module has been marked, double marked/moderated in**  **line with USW procedures.** | | |
| *Provisional mark only: subject to change and/or confirmation by the Assessment Board* | | |

# ASSESSMENT TASK - DETAILED REQUIREMENTS

**Part 1:** Develop in Renew a Petri net for the algorithm/problem assigned to you from the following list. Use the internal data structure(s) and the nets-within-nets paradigm where suitable. You will be given literature to base your model on.

1. mutual exclusion (reader/writer scenario) with and without priorities
2. broadcast algorithm
3. point-to-point messaging
4. multicast algorithm
5. dining philosophers
6. dining cryptographers
7. leader election
8. gossip
9. traffic lights
10. ATM

**Part 2:** Model a simple producer/consumer scenario in Renew, then ‘implement’ it in the C++ Petri net simulator developed in the tutorials following the example of the Finite-Automaton simulator from the lectures available on Blackboard.

*Producer/consumer scenario:* The (on-demand) producer can make a single sock at a time at a cost of £1 and has a warehouse with a capacity of 5 pairs of socks, the consumer buys pairs of socks for £4. Initially the producer has a capital of £3.

Using the simulator, show that there is a firing sequence that leads to a situation where the producer’s capital is exactly £6.

To show your understanding and to self-evaluate your solutions, provide a concise report detailing your design decisions, testing of your solution, critical evaluation (including properties of the developed nets), and an outlook.

**Deadline:** 23 January 2017, 23:59

Submit a zip archive containing all **Renew net files**, all **C++ code**, and a **report in PDF format**. Your solution may be assessed on the 30th of January 2017 in a code demo, where you will be permitted 10 minutes to demonstrate your Petri nets and application.

**NOTE**

As with all statements of requirements, the above description of the problem may contain errors, omissions and ambiguities. It is part of this coursework that you identify and resolve these, either through communication with your tutor, or through making assumptions which need to be clearly described in the work that you hand in for this task.

The nets you are constructing and the system you are building have to be your own. If you use any net models or code that is not your own, this has to be clearly stated.

**EXPECTED MODE OF WORKING**

This is an individual coursework. We need to ensure that the work you hand in **is developed by yourself.** This should not prevent you from asking for help from your tutors or from discussing elements of the work with other students.

**IF YOU EXPERIENCE ANY PROBLEMS WITH THE COURSEWORK OR HAVE A GOOD REASON WHY YOU CANNOT SUBMIT THE WORK BY THE GIVEN DEADLINE THEN PLEASE CONTACT YOUR TUTOR IN GOOD TIME.**

|  |  |
| --- | --- |
| **Part C: Reflections on Assessment**  **(to be completed by student – optional)** | |
| **Use of previous feedback:**  In this assessment, I have taken/took note of the following points in feedback on previous work: | |
| **Please indicate which of the following you feel/felt applies/applied to your submitted work**   * A reasonable attempt. I could have developed some of the   sections further.   * A good attempt, displaying my understanding and learning, with   analysis in some parts.   * A very good attempt. The work demonstrates my clear   understanding of the learning supported by relevant literature and scholarly work with good analysis and evaluation.   * An excellent attempt, with clear application of literature and   scholarly work, demonstrating significant analysis and evaluation. | |
| **What I found most difficult about this assessment:** |  |
| **The areas where I would value/would have valued feedback:** |  |

**Data Structures and Algorithms: Petri Nets**

**Introduction:**

The objective of this assignment was to produce a petri net from a given problem/algorithm using the Renew program. Whilst making use of nets-within nets to produce a more efficient design, alongside data structures. After completion of the first problem a second scenario was given to be created within the Renew program whilst also implementing the created design in C++.

**Dining Philosophers:**

The dining philosopher’s problem is a very simple problem to define. This is because the basic principal is that five philosophers sit at a table with food in the middle and having five forks between them all. The problem starts with the eating process where a philosopher requires two forks to eat as shown in Fig B, however this isn’t possible with all philosophers due to having too few forks available.

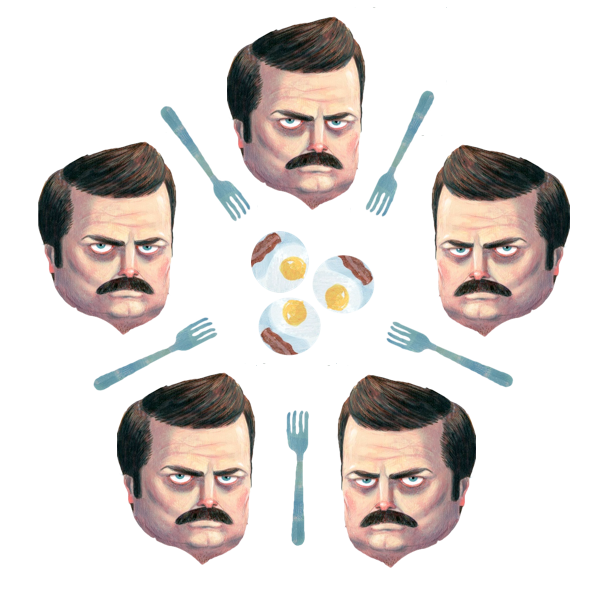


Fig A. Ron Swanson dining philosophers example (Anon, 2013)



Fig B. Philosopher requires two forks to eat not one (Anon, 2013)

Although a very simple principle, the dining philosopher’s problem has a couple issues that show off the issue of both dead-lock and live-lock. When does deadlock occur?

Deadlock occurs when every philosopher attempts to pick up a fork and will then wait for another fork to become available before they can begin eating, however none of the philosopher will drop a fork thus stuck in an eternal loop of waiting for a fork. So what about livelock? Livelock occurs when a time limit is set for each philosopher’s ability to pick up a fork and drop a fork. Although this prevents deadlock it causes another issue where the timings can synch and each philosopher drops their fork, waits out the set time and then tries to retrieve two forks. Thus ending up in a loop again.

For this assignment a simple petri net was created to demonstrate this problem as shown in fig C. This simple Petri net shows the firing sequence of picking up forks, eating and thinking. This design suffers from starvation and deadlock where a fork is unavailable and is never retrieved by a philosopher thus causing them to starve and die.

The Petri net design was first started using places for forks and then a single transition between two representing a philosopher, these philosophers then had two places representing eating and thinking in a small cycle (Fig D.) that would be able to put a philosopher into one state and then return the forks back afterwards. However, this isn’t a suitable solution due to the randomness of the fork selection causing the scenario of starvation to occur.

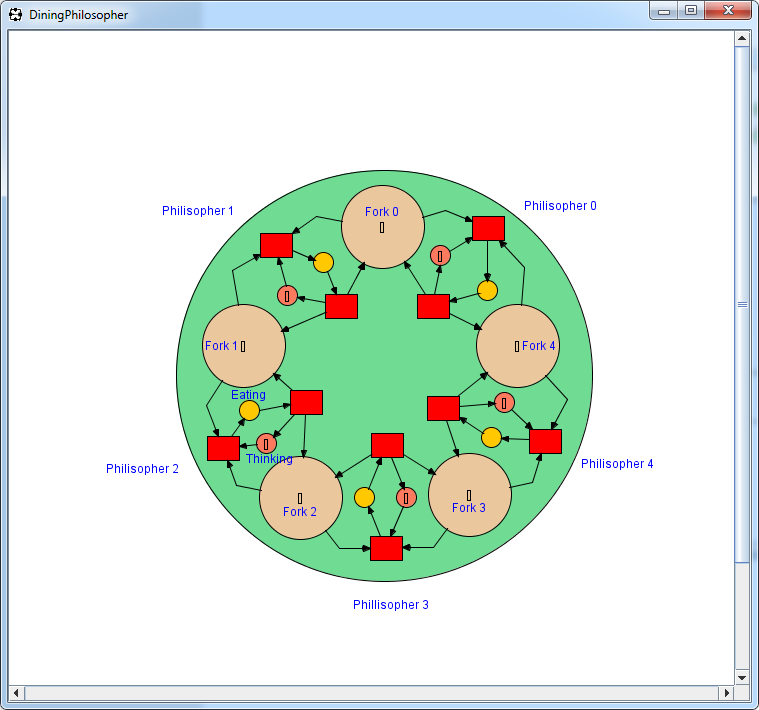
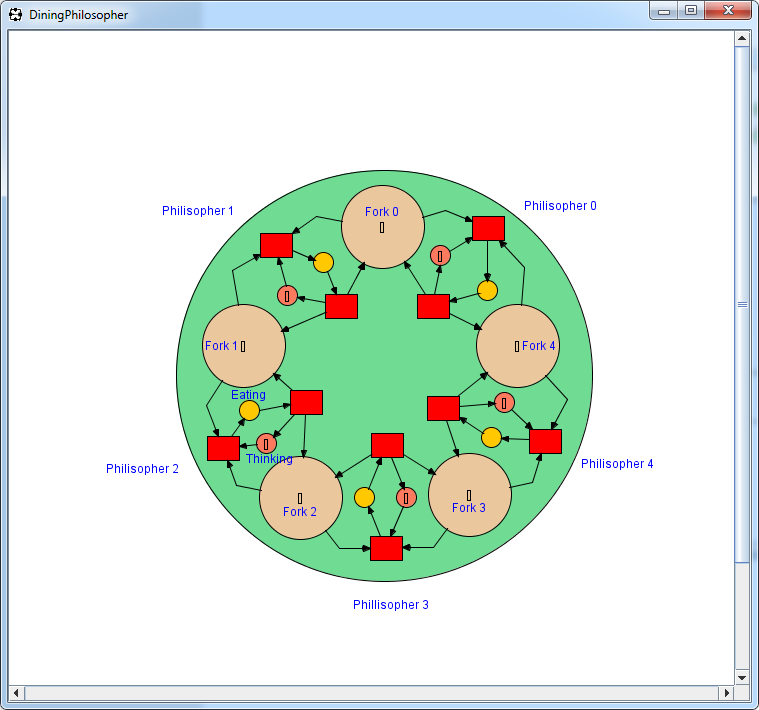
 

Fig C. Simple Philosopher Petri net. Fig D. Small philosopher cycle.

Testing of the Petri net was proven to show how the random firing has caused deadlock to occur, this is happened due to the random firing of tokens that leads to either one or more philosophers waiting a large period of time before getting the chance to eat.

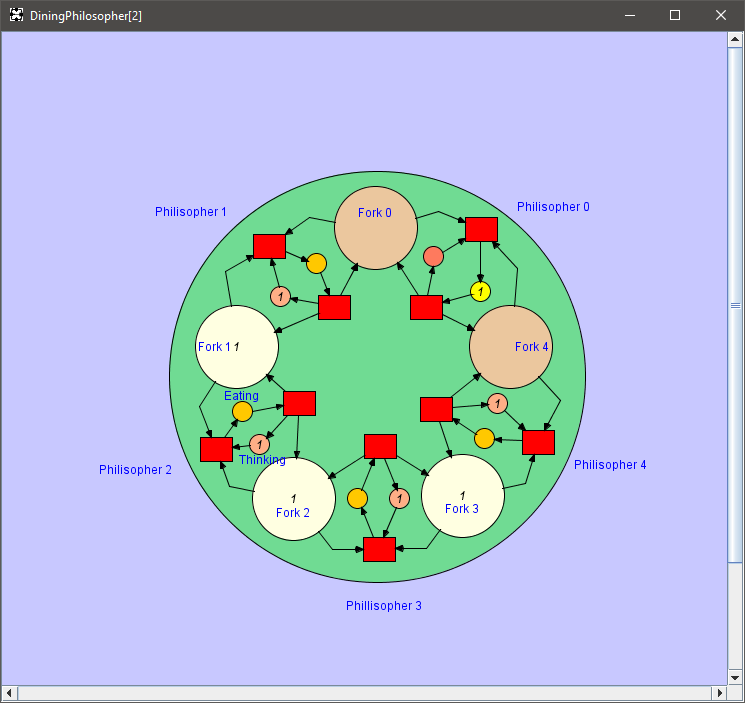


Fig E. Simple Philosopher Petri net simulation.

**Dining Philosophers Nested:**

After the first attempt of a simple petri net was created, the next step was to include nesting or nets within nets to create a slightly more complex net with a better solution to the deadlock problem. The net shown in Fig F shows a similar design to the first Petri net, although this net makes use of nets within nets by taking each philosophers states into its own net as shown in Fig G.

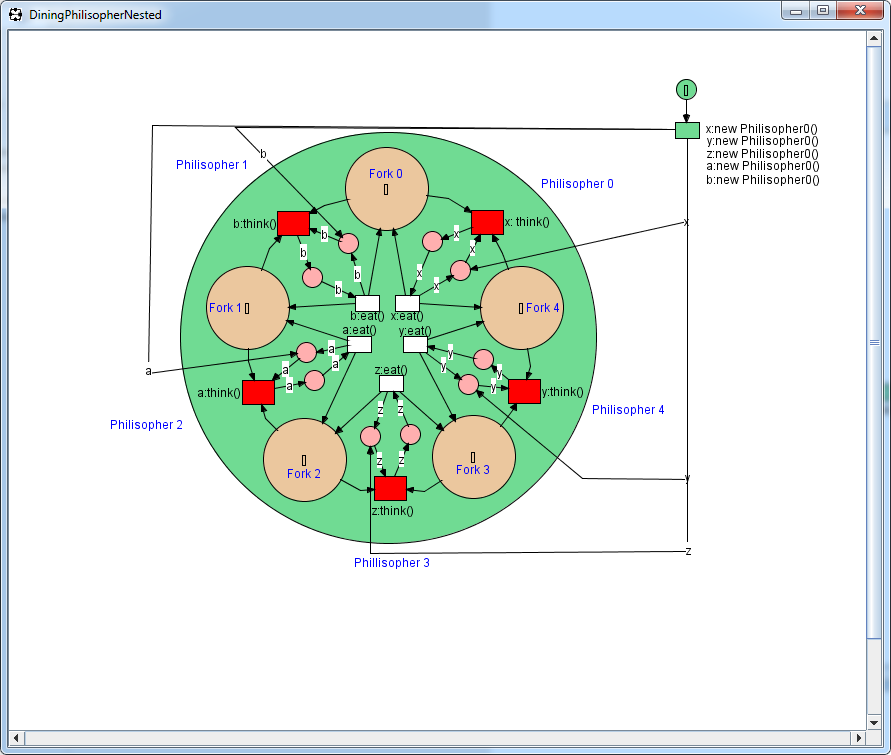


Fig F. Nested dining philosopher’s problem.

The design for this net was to move the philosopher’s states into their own separate nets and have each state as its own transition that would be able to replace tokens back for the forks without completely removing the philosopher as was previously having with the net. The net demonstrates this by having a philosopher thinking until two tokens are passed into the thinking transitions, once this has fired the philosopher will begin to eat and the tokens are placed back into the forks original place.

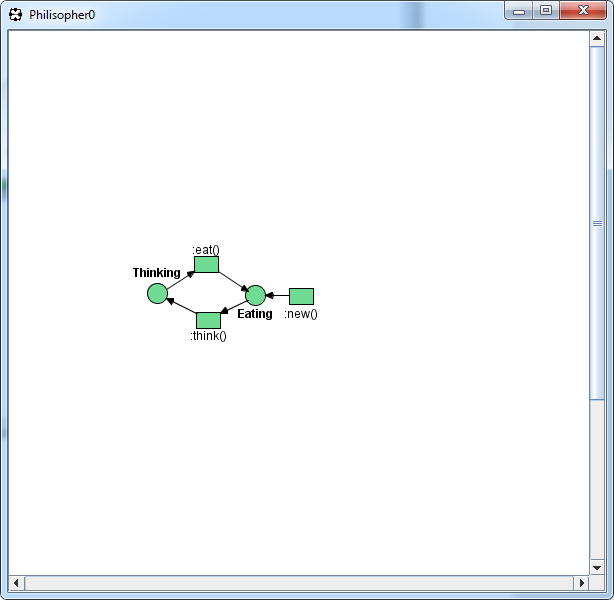
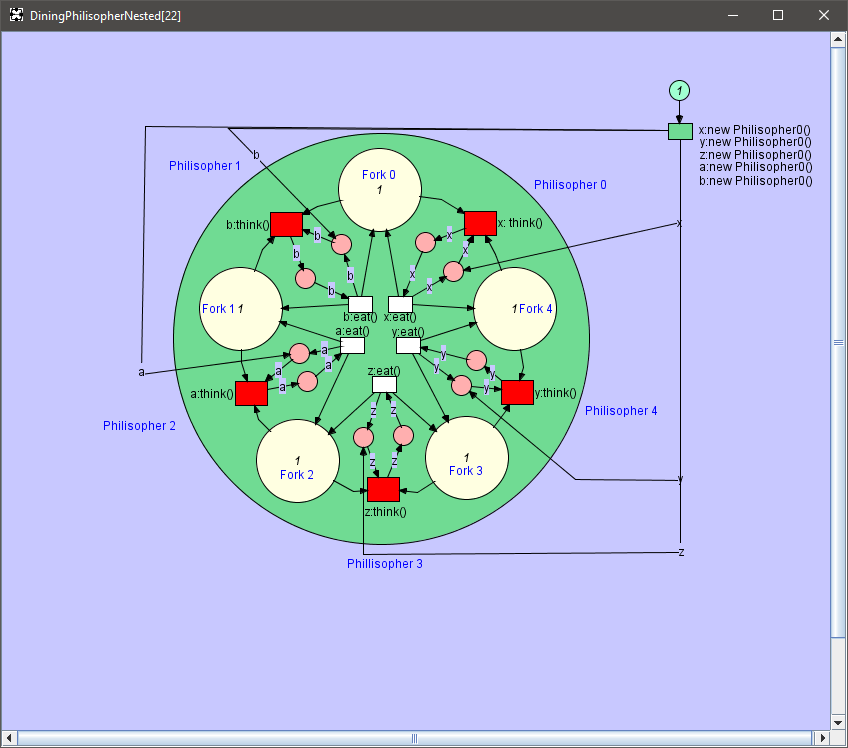
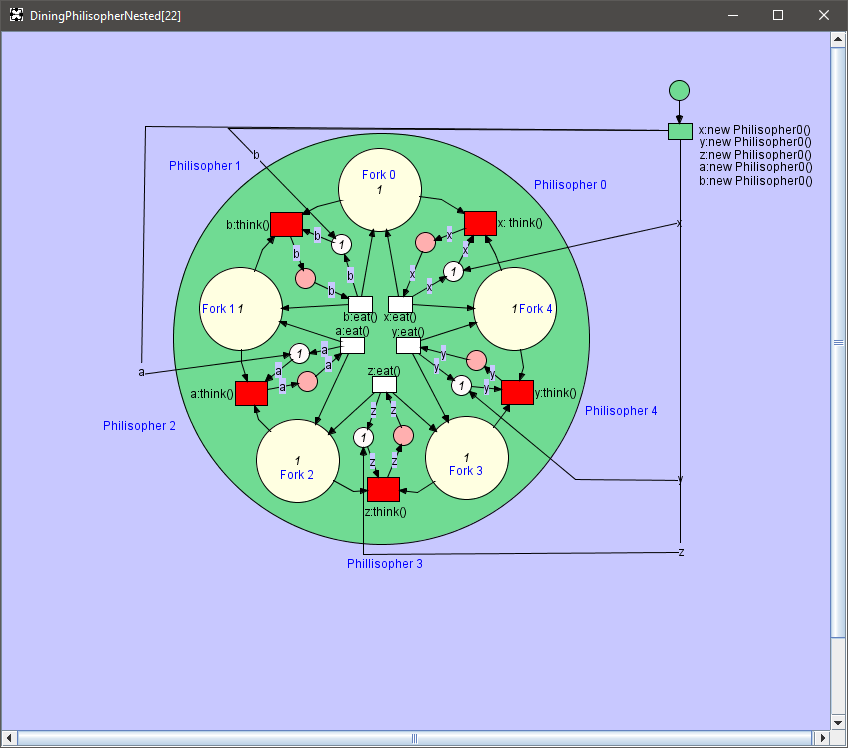


Fig G. Net within net.

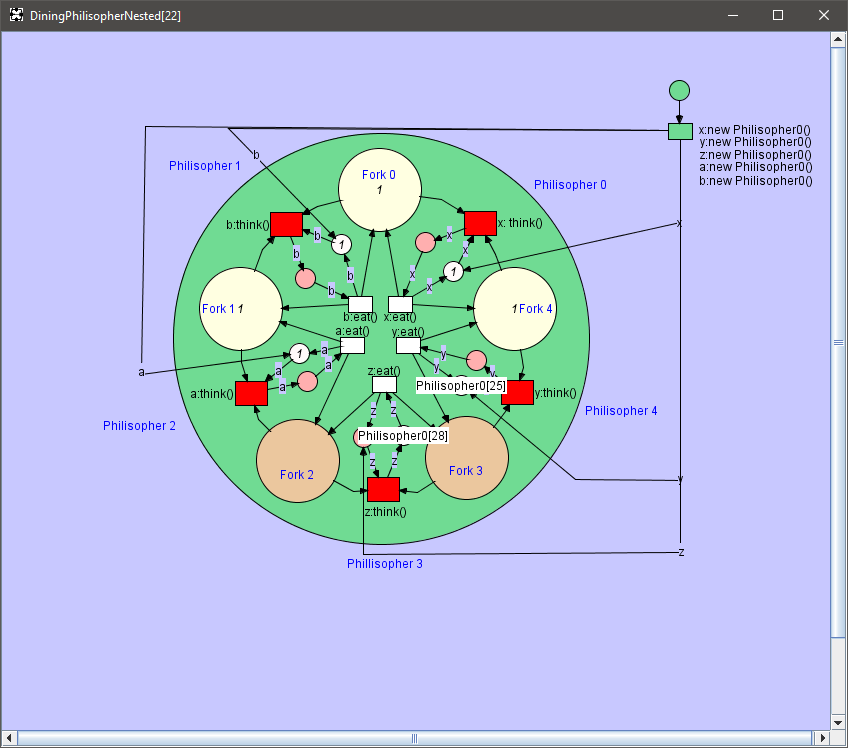
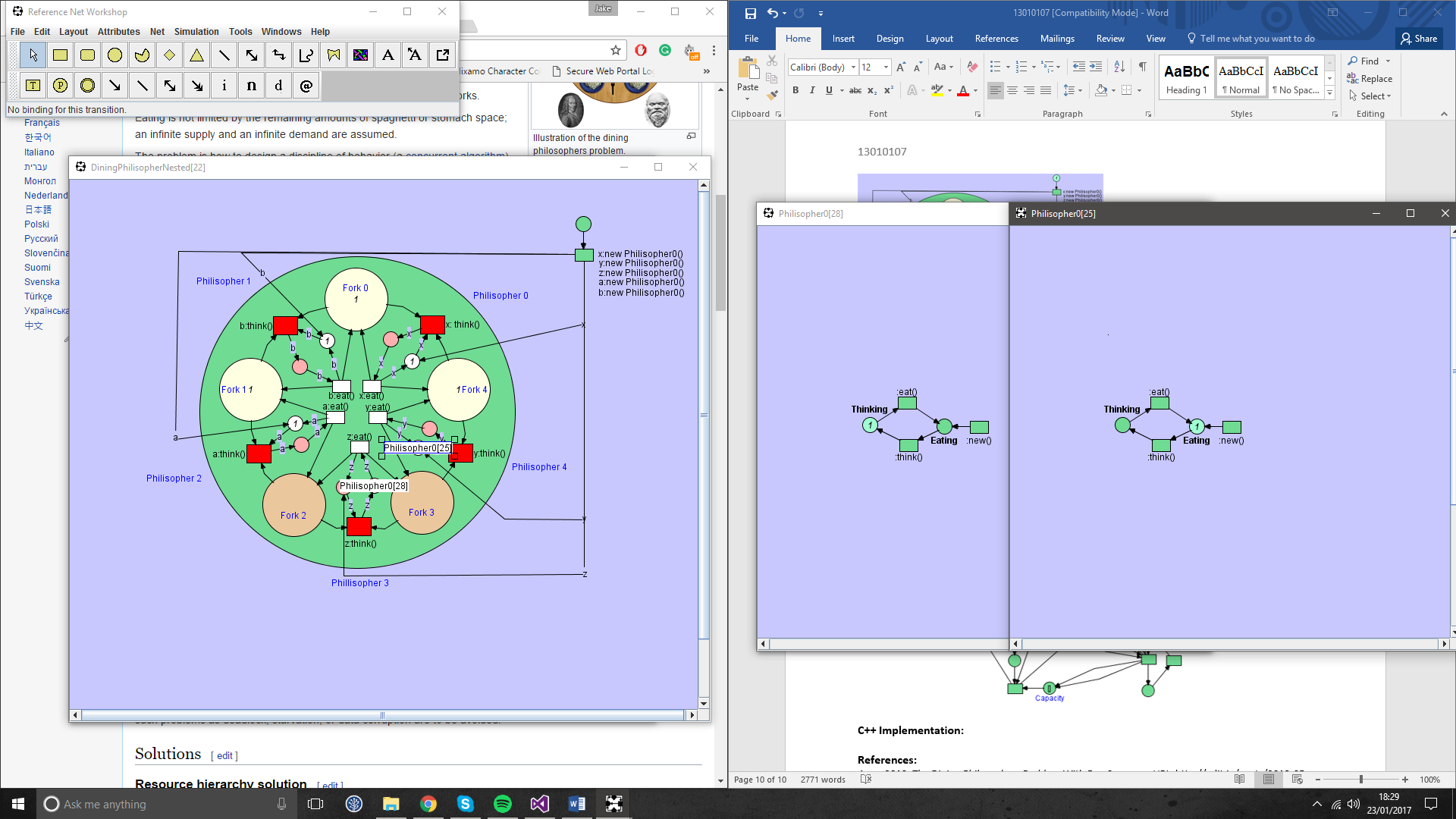
Fig G represents a very simple net with each state being fired in a cycle with the philosopher only ever being able to eat or think, not both at the same time, nor is a token ever stuck inside one state indefinitely. However, although is a smarter solution it hasn’t fixed the starvation issue due to philosopher still eating in a random order meaning one philosopher can end up waiting a large amount of time before they’ll get to eat.

Testing of this net as shown in Fig H, shows the initial firing of the Petri net. Each stage has been labelled with a number to identify the stepping of each firing sequence. As shown in stage 1 the tokens are placed into the forks ready for eating. The next stage preps the philosophers by creating a new instance of each and placing them ready in the eating stage, although this should have been the thinking stage, this is due to human error. Once they have been fired a random philosopher is chosen and begins the transition between eating to thinking, again this should be thinking to eating. With no specific exit requirement stated this process lasts until the token is sent back to eating and tokens are replaced in the forks, allowing for other philosophers to eat.



2

1



4

3

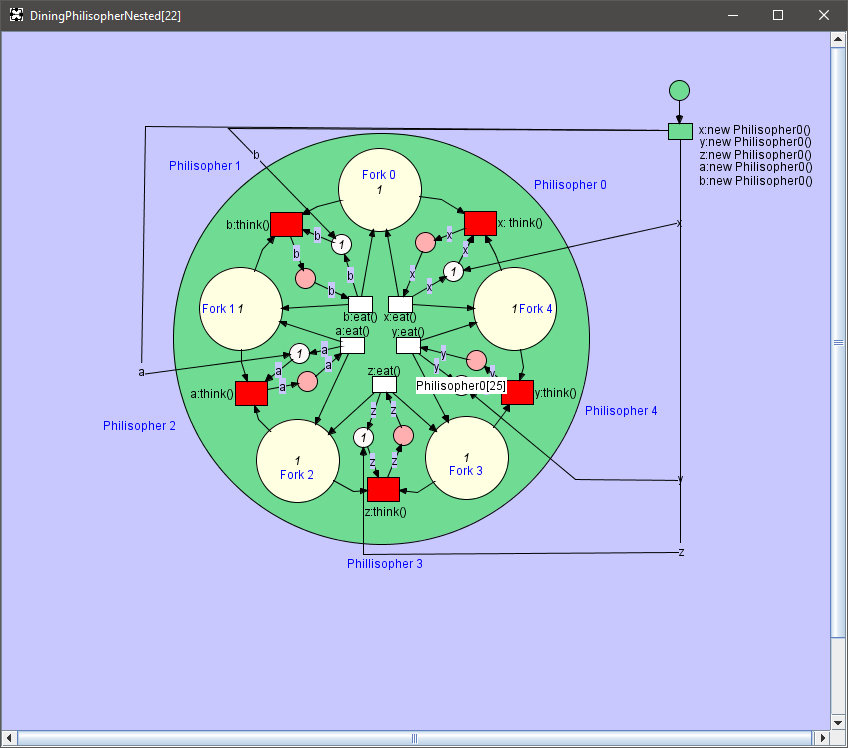
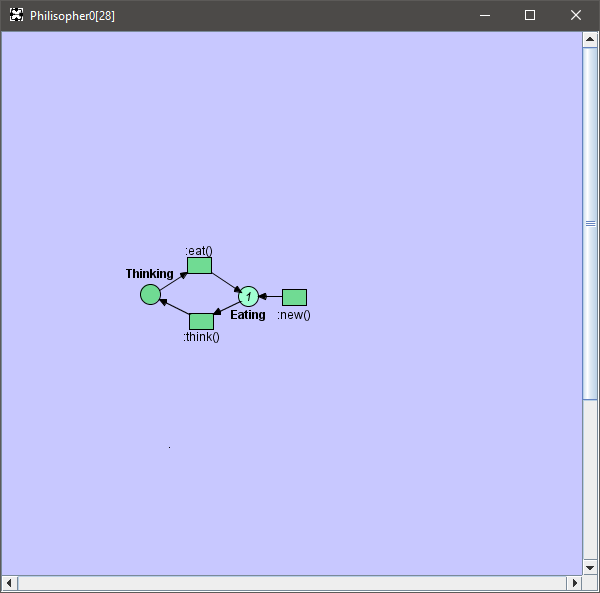


Fig H. Testing stages of the Petri net.

Although this net works a little more complex than the previous net, it still suffers from starvation and deadlock. Alongside having the thinking and eating states the wrong way round. A solution to fixing the deadlock state would be to utilise a weighted system for each fork, this creates a rule so that if you are using two forks, you would need to pick up the lowest weighted fork first. However, this system is slow, and due to time constraint s a net was not created using this solution.

**Producer/Consumer Problem:**

The second part moves onto the creation of a Petri net for a given scenario, this net was relatively simple, although required multiple arcs to simulate weightings, due to inexperience using the Renew program.

The design process for this Petri net as shown in Fig I was simple with a minimal amount of places and transitions. The first step to creating this net involved creating a sequence for producing a single sock at a time with a cost of £1, this sequence creates a token that will always loop back to the producer whilst adding a sock at the warehouse. This firing sequence was the simplest of the entire net as shown in Fig J.

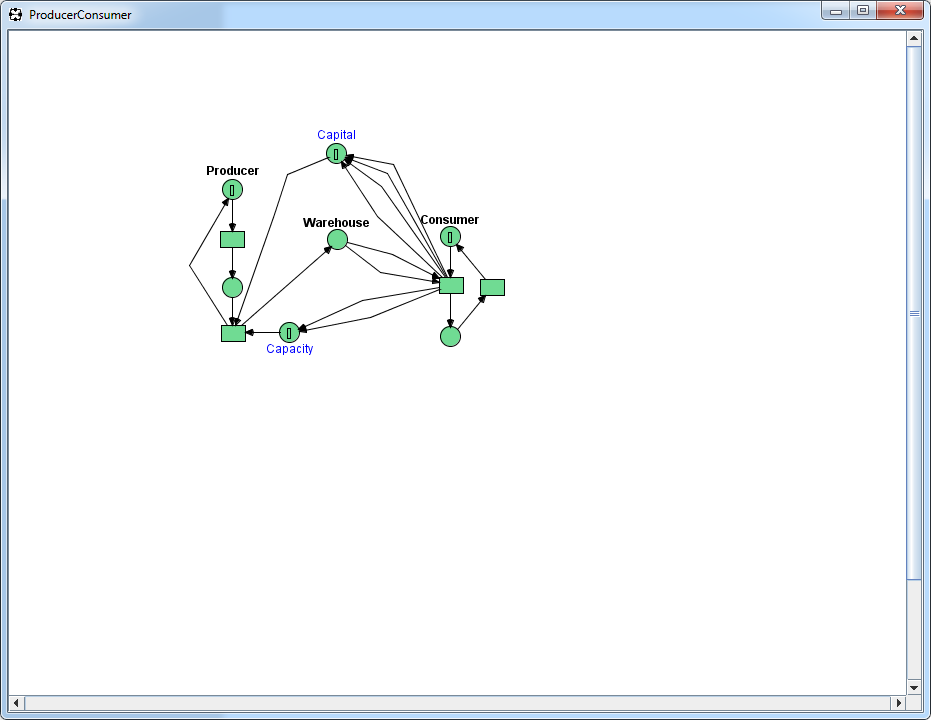
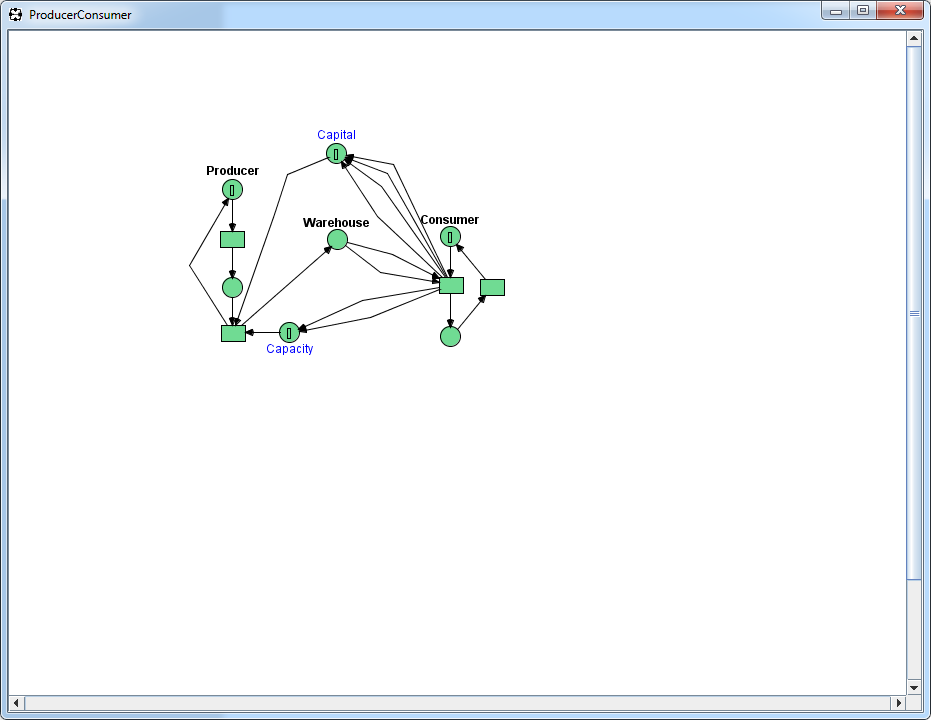
 

Fig I. Producer/Consumer Net. Fig J. Producing socks one at a time.

After producing a single sock, the net will need to move the sock into the warehouse storage which is only capable of storing 5 pairs of socks or 10 single socks. With this in mind the net was created to move socks into a separate warehouse place and then transfer to a consumer once two socks became available as a pair as shown in Fig K. However, to prevent infinite storage, once a sock goes into the storage place, a counter called capacity will only allow 10 socks/tokens to be created. This counter will increase after the socks have been moved to the consumer by using two arcs to represent two socks. Moving onto the consumer as shown in Fig L, the pair of socks are placed into the transition where they will go into the consumer once two tokens have been placed. With the consumer having received the pair of socks, £4 capital is made. This capital is placed into its own place using four arcs.

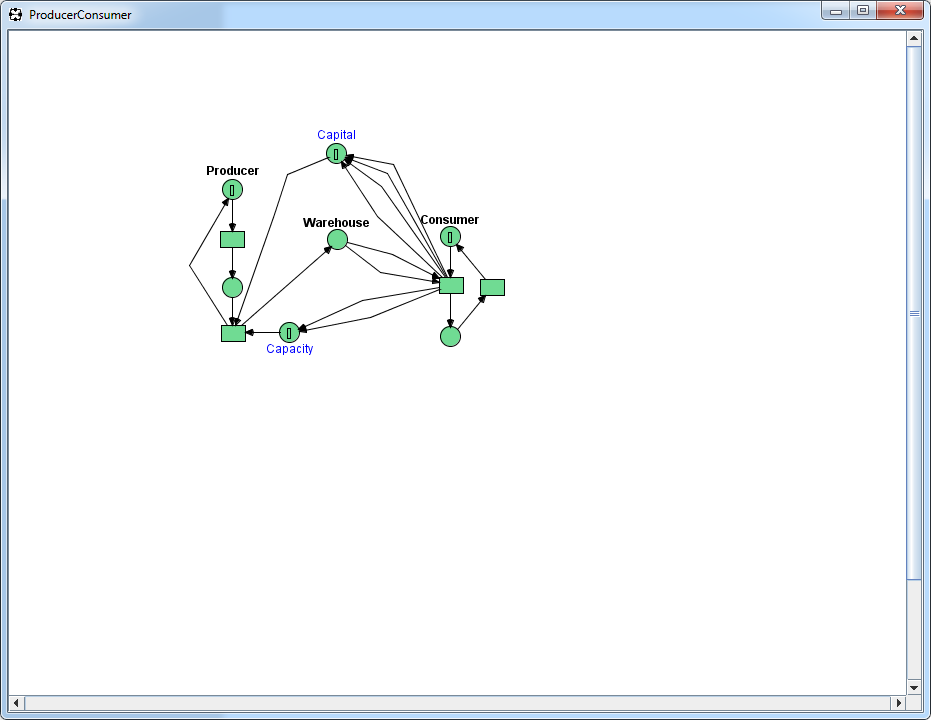
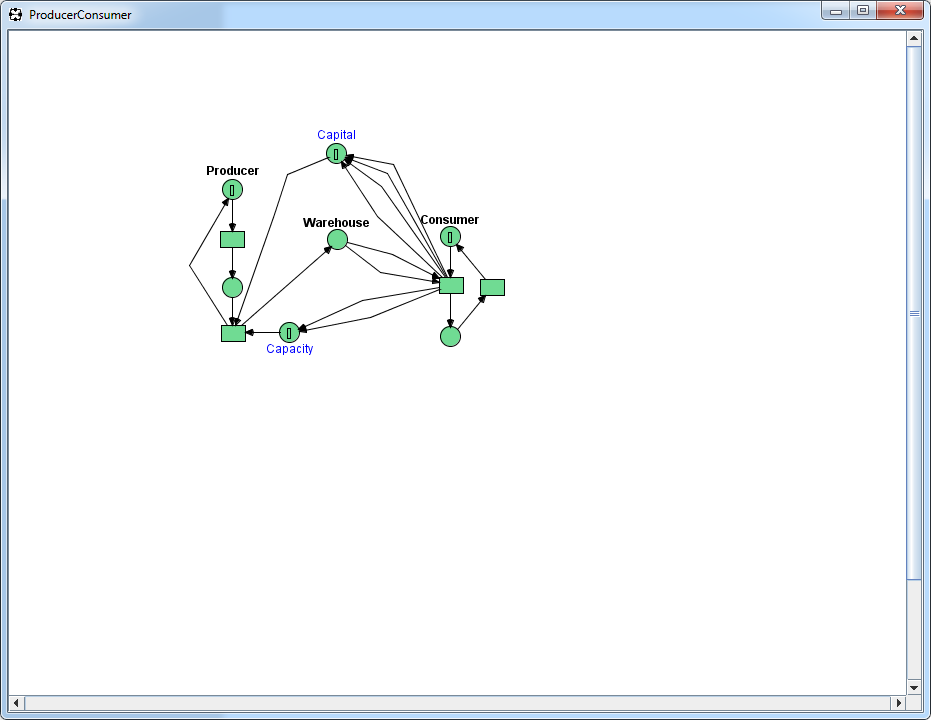
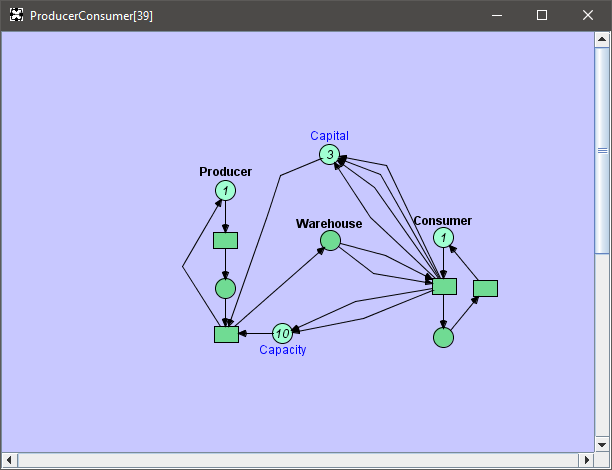
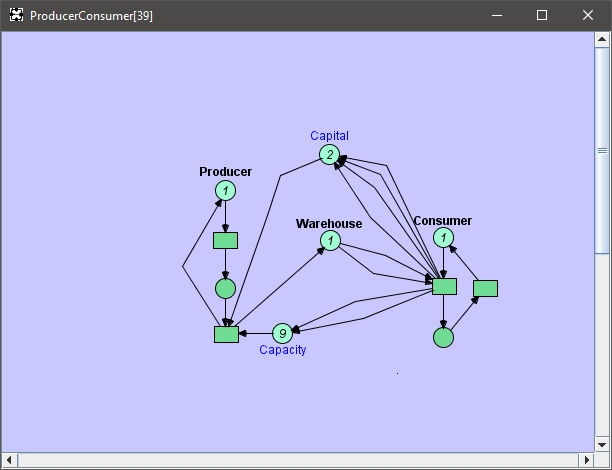
 

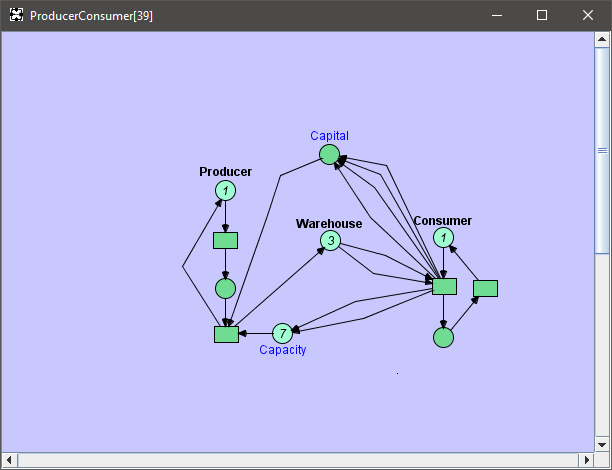
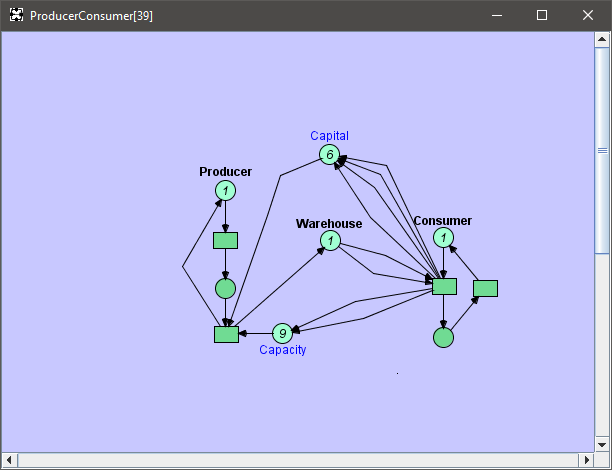
Fig K. From producer to Warehouse and then consumer. Fig L. Consumer sequence.

Testing as shown in Fig M utilising a series of image shows the movement of token from the producer to the consumer alongside the decrease of capital for producing socks at £1 each and the decrease in capacity size of the warehouse as socks are being produced. This firing sequence continues until a capital of six is visually made in image 4 of Fig M.



2

1



4

4

3

Fig M. Running stages of the Producer/Consumer scenario.

**C++ Implementation:**

After implementing the scenario within Renew, the assignment asks for an implementation of that Petri net be created within C++. The first step to creating the petri net within C++ was to create a structure header that would be used to store all the information for the Transitions, Arc and Places. This is shown in Fig N, and identifies all related variables to each state.

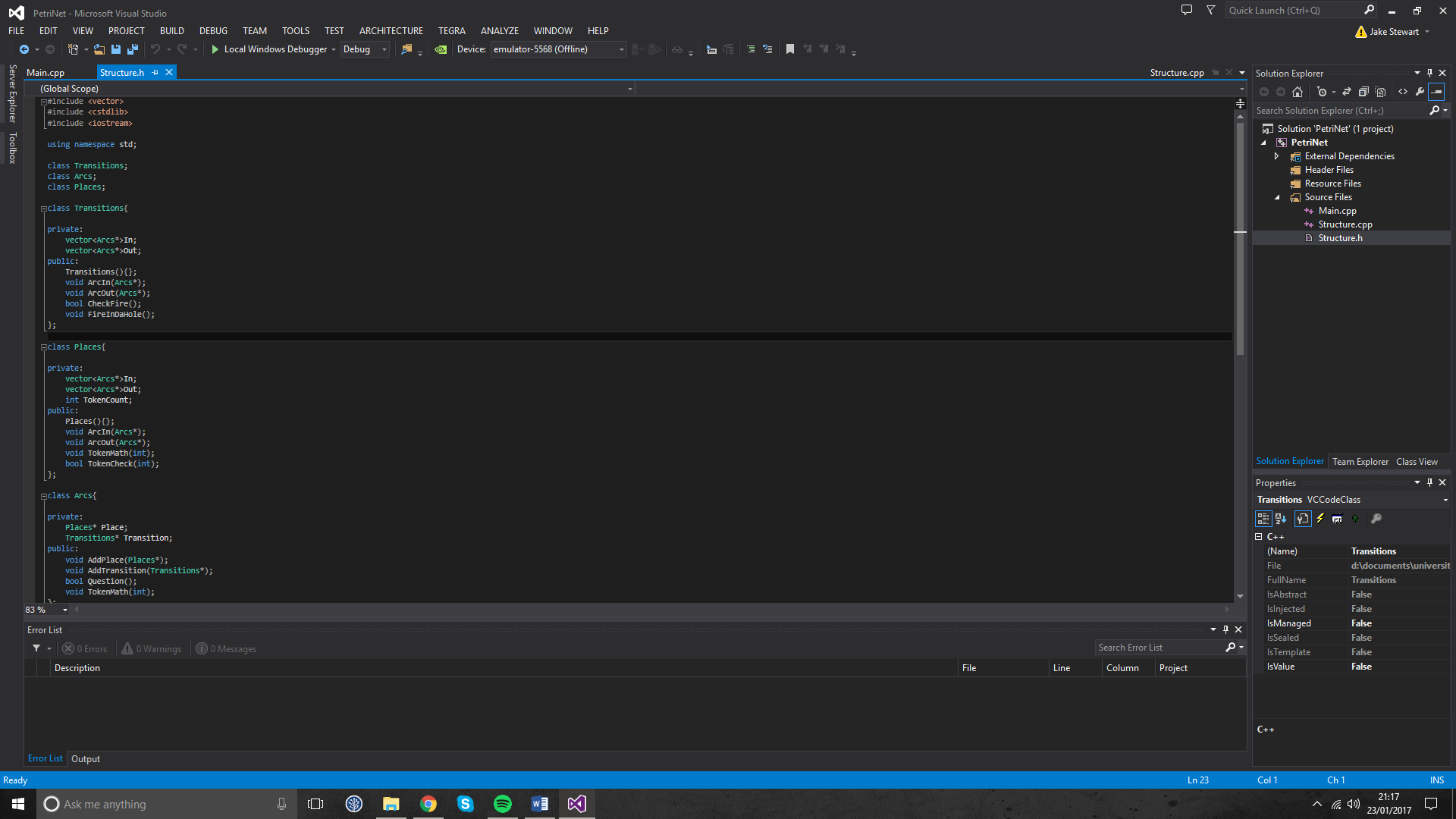


Fig N. Structures file.

The reasoning for utilising a structure header file was to easily manage all information of a function in one place and to make the use of encapsulation to protect modifiable variables that should never be touched outside of the class it’s in.

Starting on the transitions class, you can see the two private vectors In and Out. These vectors are used in the function calls ArcIn and ArcOut as an element in the array of the arcs used by the Petri net. The CheckFire function is used to check if a place has a token and whether it has enough tokens to fire. Finally, FireInDaHole is used to remove this token form the ArcIn and add one to the ArcOut from a transition. These functions can more easily be seen in Fig O, showing precisely how they go about achieving their intended use.

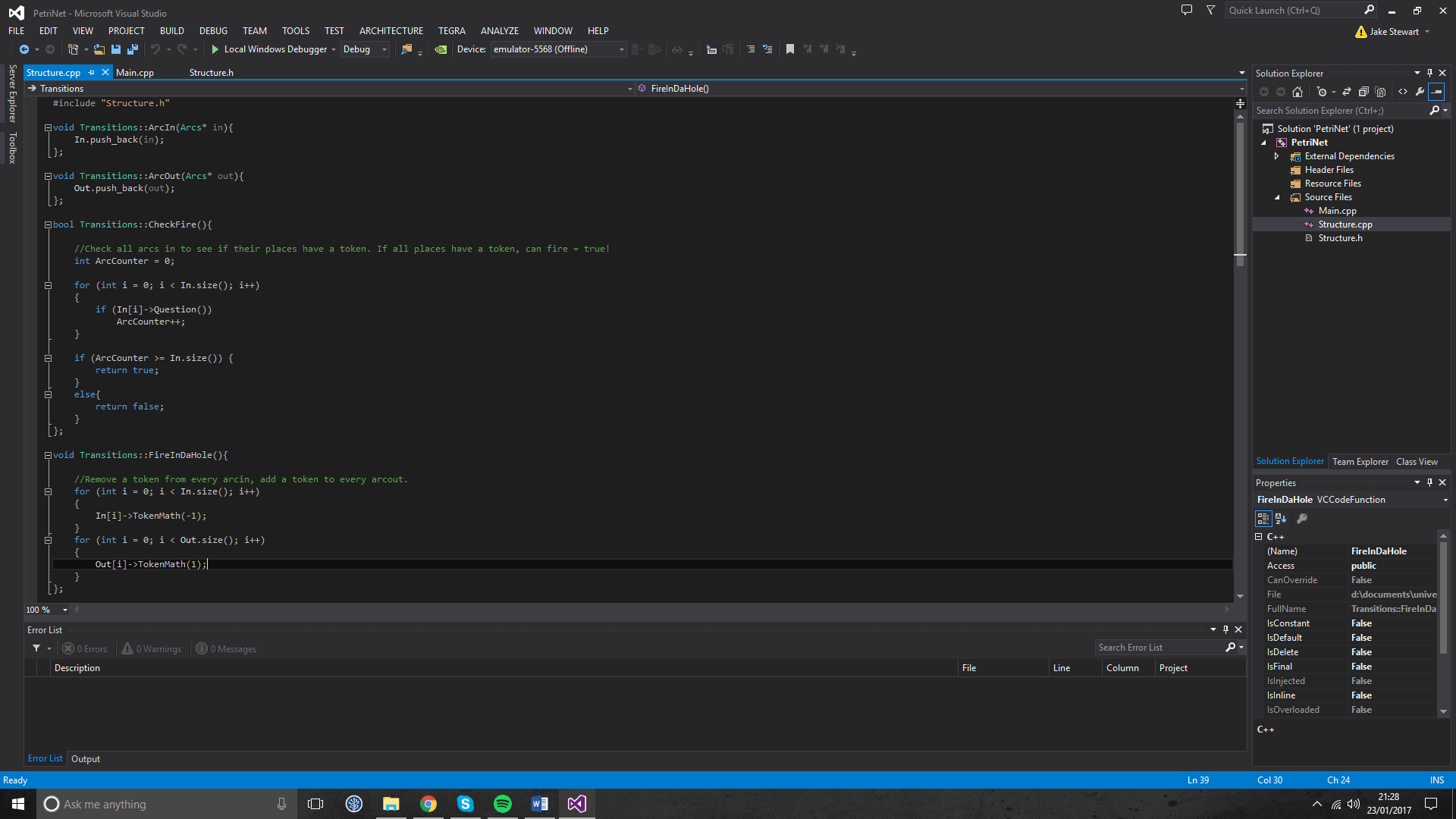


Fig O. Utilisation of the Transition Class.

The next class is of places and this also contains the information needed for Arcs, so is exactly the same as the transitions. So the only extra functions needed for a place is the ability to add or remove tokens and to check if a place has any tokens.

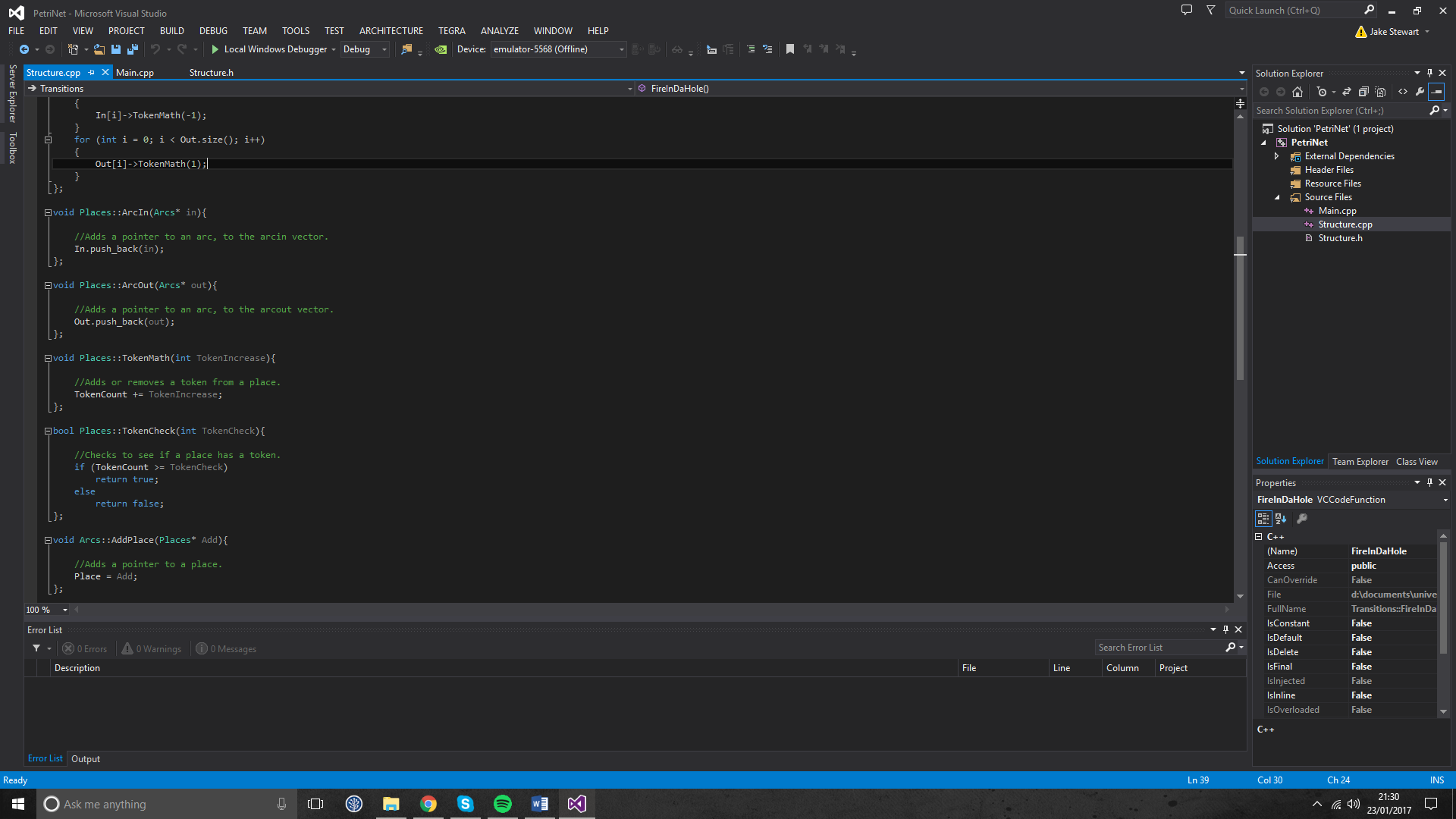


Fig P. Utilisation of the Places Class.

Finally, the last class is used for the arcs, and holds the information relating to the location of a place, the ability to add transition to a place, check if a place has any tokens that an arc is going to and then be able to remove and add tokens from a connected place.

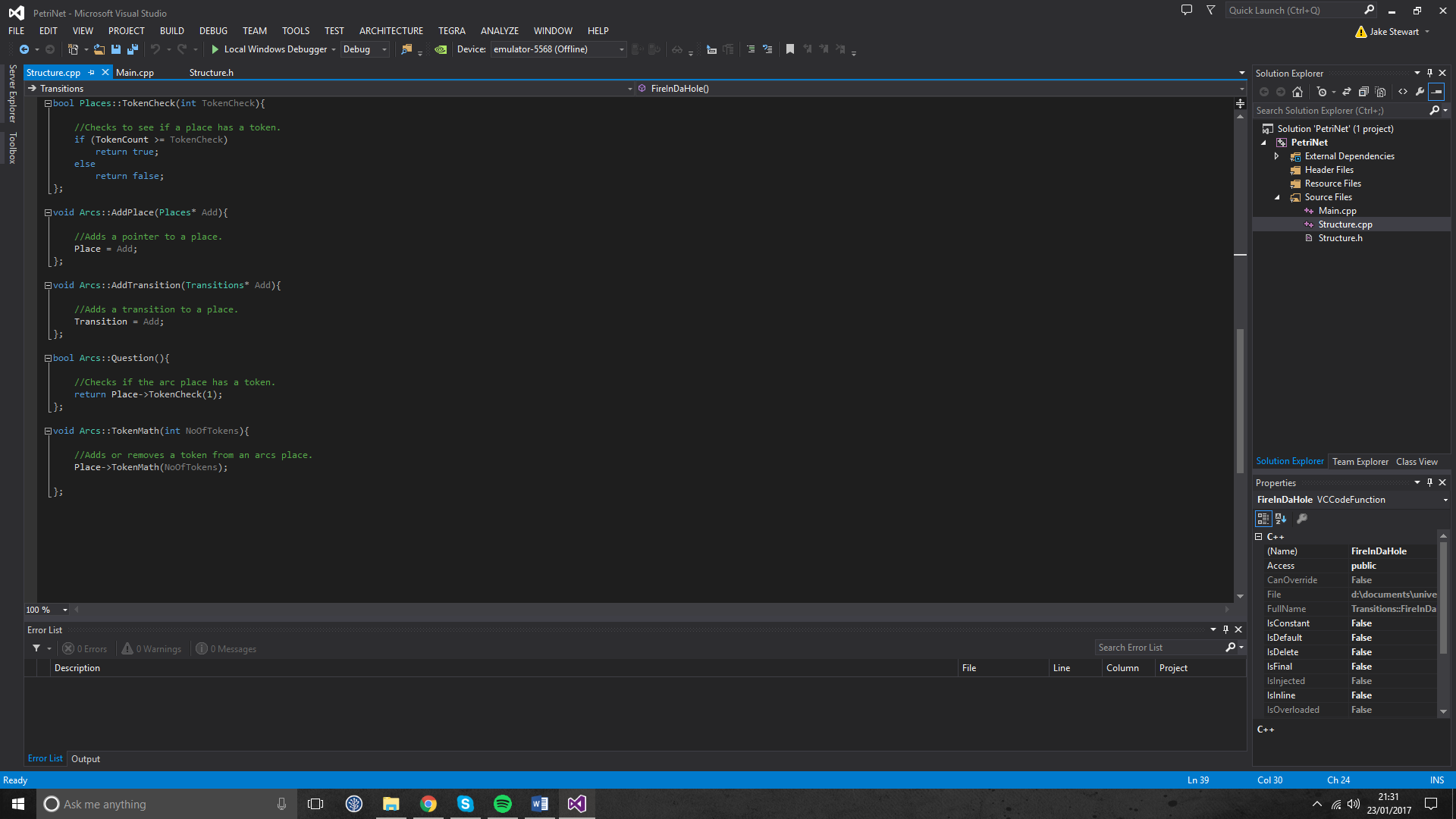


Fig Q. Utilisation of the Arcs Class.

This section of the assignment was incomplete due to time constraints and has no simulation for testing to be completed. Although the setup for a Petri net is complete, it severely lacks the code required to compile and run.

**References:**

*Anon 2013. The Dining Philosophers Problem With Ron Swanson. URL:* http://adit.io/posts/2013-05-11-The-Dining-Philosophers-Problem-With-Ron-Swanson.html

1. University Academic Misconduct Regulations [↑](#footnote-ref-1)
2. Information on exclusions to this rule is available from the Advice Centre at each Campus [↑](#footnote-ref-2)