Concurrency and Visualisation

Michael Askanis

A dissertation Submitted in partial fulfillment of the requirements for the degree of a Master of Science in Computer Science in the University of Wales, Aberystwyth

Supervisor: Dr. E M Sherratt

University of Wales, Aberystwyth

February 2004

DECLARATIONS

The content of this dissertation is the result of my own independent work and investigation except where otherwise stated. All sources are acknowledged by explicit references to the bibliography.

Signed: . . . . (Michael Askanis)

Date . . . .

I declare that this work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed: . . . . (Michael Askanis)

Date . . . .

I hereby give my consent to the dissertation, if successful, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed: . . . . (Michael Askanis)

Date . . . .

ACKNOWLEDGEMENTS

I am grateful to my supervisor Dr. E M Sherratt for her help and encouragement during the entire working time. I would also like to thank my parents for both financial and most important moral support without which I could never have completed this project.

ABSTRACT

The project investigates the subject of concurrency.

Concurrency is the illusion that the CPU is executing multiple programs or processes, but in effect is switching from one program to another so quickly that it gives the appearance that programs are running at the same time.

The study describes the 2D visualization of concurrency and the problems associated with it, using the (i) Producer/Consumer and (ii) Dining Philosophers. Visualization is very useful to students or others who wish to gain an understanding of concurrency. Two languages (Java and C/C++) were considered; the former for its graphics and the latter for its low level standard POSIX system calls. Java was chosen for its platform-independence and ability to produce web applets.

The software was successfully created with a few alterations to the original requirements. However, problems were encountered with certain Java platform dependencies.

CONTENTS

Chapter 1– Introduction 1
1.1 Aim of Project 1
1.2 Running Environment 1
1.3 Programming Language 1
1.4 Technique 2
1.5 Depiction of Chapters 2
Chapter 2 – Background 3
2.1 Processes 3
2.2 Threads 4
2.3 Many-to-One Model 5
2.4 One-to-One Model 5
2.5 Many-to-Many Model 5
2.6 UNIX 6
2.6.1 Threads 6
2.6.2 Processes 6
2.7 WINDOWS 6
2.7.1 Threads 6
2.7.2 Processes 6
2.8 The Co-operative Multithreading Model 7
2.9 The Pre-emptive Multithreading Model 7
2.10 Deadlock & Starvation 8
2.11 Deadlock prevention 9
2.11.1 Attacking Mutual exclusion 9
2.11.2 Attacking Hold and wait 9
2.11.3 Attacking No pre-emption 9
2.11.4 *Attacking Circular wait* 9
2.12 Deadlock avoidance 10
Chapter 3 – Requirements 11
3.1 Original Requirements: 11
3.1.1 The Producer/Consumer Requirements: 11
3.1.2 The Philosophers requirements: 12
3.1.3 Extra personal requirement: 13
3.2 Altered Original Requirements: 13
Chapter 4 – Methodology 14
4.1 Similarities in Methodologies 14
4.2 Risk Assessment 14
4.3 Implemented Methodology 16
Chapter 5 – Design 17
5.1 Design of Producer/Consumer Visualisation 17
5.2 Java & C implementation using JNI 19
5.3 100% Java Implementation 23
5.3.1 A Java Single Applet Design 23
5.3.2 A Three Applet Java Design 24
5.4 Producer/Consumer Class Diagrams 26

Chapter 6 – Implementation 31
6.1 The Basic Algorithm 31
6.1.1 The Bounded Buffer: 32
6.1.2 The Producer: 32
6.1.3 The Consumer: 33
6.2 Amendments to the Producer Algorithm: 34
Chapter 7 – Testing 46
7.1 Types of Testing Levels: 46
7.2 Testing Techniques 47
7.3 Program Test 48
7.3.1 *Unit Testing* 48
7.3.2 Integration (Sub-System) Testing 50
7.3.3 *System Testing* 50
7.3.4 *Regression Testing* 50
7.3.5 *Stress testing* 51
7.4 Test Cases 51
Chapter 8 – Critical Evaluation & Conclusions 55
Bibliography 58

LIST OF FIGURES

Figure 1 – The life of a Thread 5
Figure 2 – Deadlock...8
Figure 3 – Fowler’s Incremental Software Development Process..............................15
Figure 4 – The Producer/Consumer GUI look..17
Figure 5 – The Main Menu Component 18
Figure 6 – The Main Buttons Component 18
Figure 7 – The Main Animation Component..18
Figure 8 – The Control Panel Component 18
Figure 9 – The Animation Path of the Item. 19
Figure 10 – Java and C Interfacing using JNI 21
Figure 11 – 100% Java class implementation...24
Figure 12 – The three JApplet Mechanism...25
Figure 13 – Producer/Consumer Concept...31
Figure 14 – Final Implemented Producer Algorithm...44

LIST OF CODE SAMPLES

Code Sample 1 – C POSIX Standard system calls 23
Code Sample 2 – Amended Producer Code...36
Code Sample 3 – Synchronization in Java...37
Code Sample 4 – The Animation Sequence 40
Code Sample 5 – Infinite thread loop 41
Code Sample 6 – The Thread Termination Technique...42
Code Sample 7 – The Thread Activation, De – Activation Technique 43
Code Sample 8 – The original timer method..49
Code Sample 9 – The amended Timer Method 49
Code Sample 10 – Determining the Operating System 57

LIST OF CLASS DIAGRAMS

Class Diagram 1 – MainAnimationApplet.java..27
Class Diagram 2 – BoundedBuffer.java 28
Class Diagram 3 – Semaphore.java 28
Class Diagram 4 – Producer.java...29
Class Diagram 5 – Consumer.java...29
Class Diagram 6 – ControlPanel.java 29
Class Diagram 7 – MainButtons.java 30
Class Diagram 8 – MainMenu.java 30

LIST OF TABLES

Table 1 – Item range coordinates relative to the Window 39
Table 2 – Animation Panel Test Case..52
Table 3 – Main Menu Test case..53
Table 4 – Main Buttons Test Case..53
Table 5 – Control Panel Test Case..54

Chapter 1- Introduction

The purpose of this chapter is to describe briefly the aim of this project. In addition it outlines the context of each chapter of which the dissertation includes.

1.1 Aim of Project

The aim of the project is to visualize concurrent systems, i.e. processes which operate in apparently the same time, from within a single computer system with a single CPU. The motivation is to demonstrate the problems that arise when processes that communicate with each other through a shared resource(s) do not do so in a synchronized manner.

The user interface and presentation of this problem with the use of 2D visualization allows the user(s) to visually witness the problems and also give them the facility to make adjustments through a menu and control panel.

1.2 Running Environment

The implementation of the code was originally intended to be targeted under the SOLARIS UNIX and variant platforms. It was later changed to include Windows environments as well, which led to the use of 100% Java programming as opposed to a mixture of both Java and C. This meant that the software would run in theory on any UNIX clone and any Windows version. In practice, incompatibilities and inconsistencies arose with the two Operating Systems existing on campus, SOLARIS 2 and Windows XP.

1.3 Programming Language

The initial design intended to use a mixture of Java and C, the former for its graphics and the latter for its low level standard POSIX system calls. JNI would act as the translator between the two programming languages. POSIX standards were part of the initial Requirements. This later changed during the Design/Implementation to 100% Java standards due to experienced problems in SOLARIS 2 with the use of JNI. C alone could not be used due the time constraints requiring to self learn the X11/Motif package libraries.

1.4 Technique

The subject of Concurrency was new to me. The approach I chose was to design, implement and test in an incremental manner. Experimenting in this way gave me the opportunity to see what was feasible in terms of practical programming and also gave me ideas of what graphically to show in my application. In general, the dissertation was more practical, my individual theoretical concepts were expanded with the experimentation of programming and explaining the output phenomena by researching with books or the internet.

1.5 Depiction of Chapters

The dissertation is structured as follows:

- **Chapter 2 -** Explains the subject of Concurrency and its relation to various operating systems on computers with a single CPU. How this is achieved, elaborating the benefits and also the problems that may arise.

- **Chapter 3** - Provides a list of requirements, functions and details for the process of constructing the software system. These requirements set out what the software package should do, and define constraints on its operation and implementation.

- **Chapter 4** - Describes briefly several possible methodology principles, similar phases which exist between them and finally which of those described was chosen as opposed to the others for the development of this project's software according to the requirements outlined in Chapter 3.

- **Chapter 5 -** Describes the main design of the UI of the Software and the various implementation approaches that could be used to accomplish it. With each implementation approach its corresponding problems and risks are described.

- **Chapter 6 -** Explains the implementation of the successful chosen design developed in Chapter 5 and elaborates on certain interesting algorithms used to accomplish the requirements specified in Chapter 3.

- **Chapter 7 -** Describes the strategy used to produce and carry out test cases which would be used to see if the requirements outlined in Chapter 3 were fulfilled accurately.

- **Chapter 8 -** Outlines the positive and negative aspects of the project that occurred during the process of the development. An evaluation of the final product is done to see if the requirements were fulfilled as planned.

- **Bibliography**

Chapter 2- Background

The purpose of this Chapter is to explain the subject of Concurrency and its relation to various operating systems on computers with a single CPU. How this is achieved, elaborating the benefits and also the problems that may arise.

The CPU (Central Processing Unit) is the heart of any computer, but the operating system is the brain which [1, 2, 5, 6].

- Manages the hardware and software resources of the computer system. These resources include such things as the processor, memory, disk space, etc.
- Provides a stable, consistent way for applications to deal with the hardware without having to know all the details of the hardware.

Managing the hardware and software resources is very important, as various programs compete for the attention of the CPU and demand memory, storage and I/O for their own purposes. In this capacity, the operating system makes sure that each application gets the necessary resources in turn without any interference.

The second task is to provide a consistent application interface. A consistent application program interface (API) allows a software developer to write an application on one computer and have a high level of confidence that it will run on another computer of the same type, even if the hardware specifications of the other machine for example, amount of memory or the quantity of storage is different on the two machines. Even if a particular computer is unique, an operating system can ensure that applications continue to run when hardware upgrades and updates occur, because the operating system and not the application is charged with managing the hardware and the distribution of its resources

Managing the processor comes down to two related issues.

- Ensuring that each process and application receives enough of the processor's time to function properly.
- Using as many processor cycles for real work as is possible.

2.1 Processes

A Process has the following characteristics [1, 2, 5, 6].

- Each process has its own separate program space.
- Process A cannot read or write into process B's program space.
- Each process carries the same overhead in terms of bulk that an EXE or executable file requires.

The operating system arranges the execution of applications so that it seems that there are several things happening at once. The CPU can only do one thing at a time. In order to give the appearance of lots of things happening at the same time, the operating system has to switch between different processes thousands of times a second.

- A process occupies a certain amount of RAM. It also makes use of registers, stacks and queues within the CPU and operating-system memory space.
- When two processes are multi-tasking, the operating system allots a certain number of CPU execution cycles to one program.
- After that number of cycles, the operating system makes copies of all the registers, stacks and queues, used by the processes and notes the point at which the process paused in its execution.
- It then loads all the registers, stacks and queues, used by the second process and allows it a certain number of CPU cycles.
- When those are complete, it makes copies of all the registers, stacks and queues, used by the second program and loads the first program.

All of the information needed to keep track of a process when switching is kept in a data package called a process control block which contains the following.

- An ID number that identifies the process.
- Pointers to the locations in the program and its data where processing last occurred.
- Register contents.
- States of various flags and switches.
- Pointers to the upper and lower bounds of the memory required for the process.
- A list of files opened by the process.
- The priority of the process.
- The status of all I/O devices needed by the process.

When the status of the process changes, from pending to active, for example, or from suspended to running, the information in the process control block must be used like the data in any other program to direct execution of the task-switching portion of the operating system. This process swapping happens without direct user interference and each process gets enough CPU cycles to accomplish its task in a reasonable amount of time.

2.2 Threads

A thread is similar to a process in that a thread and a running program are both a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a complete program and takes advantage of the resources allocated for that program and the program's environment [14]. This allows it to be more memory efficient because it can be contained within a single executable. The Operating System Kernel does not have to copy the entire message from one program space to another which allows it to run faster because switching between Threads does not take the same amount of workload as a process. As a sequential flow of control, a thread must carve out some of its own resources within a running program this entails having its own execution stack and program counter. On the other hand Threads are typically not loadable. That is, to add a new thread, you must add the new thread to the source code, then compile and link to create the new executable. Processes are loadable, thus allowing a multi-tasking system to be characterized dynamically. For example, depending upon system conditions, certain processes can be loaded and run to characterize the system. However, the same can be accomplished with threads by linking in all the possible threads required by the system, but only activating those that are needed, given the conditions. Threads can walk over the data space of other threads. This cannot happen with processes. If an attempt is made to walk on another process an exception error will occur. The following diagram shows the states that a Java thread can be in during its life. It also illustrates which method calls cause a transition to another state. This figure is not a

complete finite state diagram, but rather an overview of the more interesting and common facets of a thread's life.

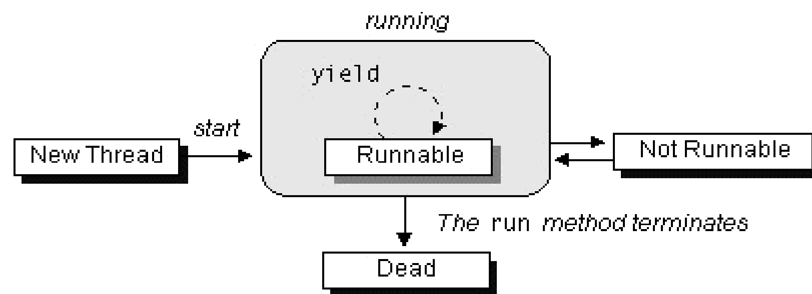


Figure 1 – The life of a Thread

(From: Sun Microsystems [9])

2.3 Many-to-One Model

This technique maps many user-level threads to one kernel thread. Thread management is done in user space, so it is efficient. The entire process will block if a thread makes a blocking system call. Multiple threads are unable to run parallel on multiple CPU's because only one thread can access the kernel at any one time [8].

2.4 One-to-One Model

This technique maps each user thread to a kernel thread. In comparison with the Many-to-One Model it provides better concurrency in that it allows another thread to run when a blocking system call is created and also threads can run on multiple CPU's. But because each thread has a corresponding Kernel thread a burden is created on the application causing restrictions on the number supported by the system [8].

2.5 Many-to-Many Model

This technique connects the user level threads to an equal or less number of Kernel threads. A User-level threads library provides sophisticated scheduling of user-level threads above kernel threads. The kernel needs to manage only the threads that are currently active. By removing restrictions on the number of threads that can effectively be used in an application there is less programming effort. A many-to-many multithreading implementation thus provides a standard interface, a simpler programming model, and optimal performance for each process [8].

2.6 UNIX

2.6.1 Threads

Threads in modern UNIX environments are managed and structured according to the POSIX.1c standard [3, 4]. When a thread is created in UNIX, it stores its execution stack, program counter value, register set and thread state (active, dead, etc.). Each thread accesses the same block of memory, so if there is only one processor only one thread can run at a time. Threads can simulate multiprocessor behaviour using mutexes or semaphores. Having multiple threads within a process and switching from thread to thread, however, takes much less time than having multiple processes switching from process to process (called *context switching*) [1, 2, 5, 6].

2.6.2 Processes

When a program is executed on a UNIX system and memory is allocated for its execution, the program becomes a process. Processes are managed by the operating system, and are structured hierarchically (i.e. processes have parent-child relationships) . Each running process can have one or more threads, which are sequences of instructions running one after another. In a multiprocessor system, each of these threads can be assigned to a different processor, so that a single process (or running program) can be handled by multiple processors by dividing its threads among them.

2.7 WINDOWS

2.7.1 Threads

When a Windows thread is created, it stores a thread ID, two execution stacks and a storage area that other programs can use to store information within the thread. In addition, the thread objects themselves have many attributes and methods that can be accessed programmatically. Attributes include execution time, priority, and exit status. Methods (or services) include thread creation, access, and termination.

2.7.2 Processes

Each process is implemented as a Windows object, which means it can take advantage of object properties and services. Unlike UNIX, Windows stores each process object as a separate entity, and there is no relationship between processes. Each process in Windows can, as in UNIX, have one or more threads that can run simultaneously on multiple processors if they exist, or share memory allocation if there is only one processor. In Windows, however, each process must have at least one running thread. This is because the thread actually contains the executable program. Each process in Windows is allocated its own memory by Windows' virtual memory manager. For each thread within a process, a unique thread object is created that maintains that thread's attributes. This differs from how UNIX handles thread attributes, in that UNIX stores pointers to attribute structures, and multiple threads can point to the same attribute [16].

2.8 The Co-operative Multithreading Model

In a cooperative system, a thread retains control of its processor until it decides to give it up (which might be never). The various threads have to cooperate with each other or all but one of the threads will be "starved" (meaning that they will never be given a chance to run). Scheduling in most cooperative systems is done strictly by priority level. When the current thread gives up control, the highest-priority waiting thread gets control. An exception to rule is Windows 3.x, which uses a cooperative model but doesn't have a good scheduler and therefore the window that has the focus gets control [17].

The main advantage of cooperative multithreading is that it's very fast and has a very low overhead. A context swap, a transfer of control from one thread to another, can be performed entirely by a user-mode subroutine library without entering the OS kernel. In NT, which is a worst-case example, entering the kernel wastes approximately 600 machine cycles [16]. A user-mode context swap in a cooperative system does little more than a C set jump/long jump call would do. Thousands of threads can be in an application significantly impacting performance. Since loosing control involuntarily in cooperative systems is not an issue, therefore synchronization is neither. That is, a programmer need not worry about an atomic operation being interrupted. The main disadvantage of the cooperative model is that it's very difficult to program cooperative systems. Lengthy operations have to be manually divided into smaller chunks, which often must interact in complex ways.

2.9 The Pre-emptive Multithreading Model

In a pre-emptive model, a sort of timer is used by the operating system itself to cause a context swap. The interval between timer ticks is called a time slice. Pre-emptive systems are less efficient than cooperative ones because the thread management must be done by the operating-system kernel, but it is easier to program with the exception of synchronization issues and tend to be more reliable since starvation is less of a problem. The most important advantage to pre-emptive systems is parallelism. Since cooperative threads are scheduled by a user-level subroutine library, not by the OS, the best a programmer can get with a cooperative model is concurrency. To get parallelism, the OS must do the scheduling. Of course, four threads running in parallel will run much faster than the same four threads running concurrently.

Some operating systems, like Windows 3.1, only support cooperative multithreading. Others, like NT, support only pre-emptive threading. (You can simulate cooperative threading in NT with a user-mode library like the "fibres" library [16]. But fibres are not fully integrated into the OS.) Solaris provides the best (or worst) of all worlds by supporting both cooperative and pre-emptive models in the same program.

2.10 Deadlock & Starvation

2.10.1 Deadlock

Deadlock can be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other.

For deadlock to exist the conditions below must all exist, without all of these, a deadlock is not possible [1, 2, 5, 6]:

1. **Mutual Exclusion** At least one resource must not be shareable.
2. **Hold and Wait** At least one process must be holding a resource and waiting to acquire a resource currently held by another process.
3. **No Pre-emption** Resources cannot be taken away from a process; it must release them as a normal event.
4. **Circular Wait** A process holding one resource (A) and waiting for another (B) must be matched by another process that needs A and holds B.

Normal deadlock occurs when two or more processes are blocking each other in a cycle of granted and blocked lock requests. For example, say Process P1 has a lock on Resource R1 and is blocked waiting for a lock on Resource R2 held by Process P2. Process P2 has a lock on Resource R2 and is blocked waiting for a lock on Resource R3 held by Process P3, and Process P3 has a lock on resource R3 and is blocked waiting for a lock on Resource R1 held by Process P1. This scenario is illustrated in the following figure.

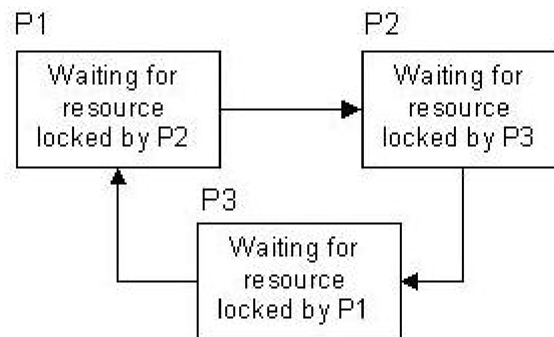


Figure 2 – Deadlock

2.10.2 Starvation

Starvation can be defined as the indefinite waiting of a process due to another process using the requested resources. Therefore deadlock implies starvation but not vice versa.

2.11 Deadlock prevention

A deadlock is not desirable therefore some methods exist to prevent deadlocks. Simple strategies are available to prevent the occurrence of a deadlock with each of the above defined four conditions. These are as follows [1, 2, 5, 6].

2.11.1 Attacking Mutual exclusion

In general, this condition cannot be disallowed. For example, it is not realistic for two processes to print on a same sheet of paper at the same time.

2.11.2 Attacking Hold and wait

This condition is preventable since a process may request for the resources it needs all at once. If its need can be met, then the operating system just does it, otherwise blocks the process until all the resources are available. However this method raises two problems listed below.

• Processes may need some resources for part of their lives. It is inefficient for a process to own all the resources all the time.
• In some cases, it is impossible for a process to know in advance what resources it will need during its execution.

2.11.3 Attacking No pre-emption

This condition can be prevented in two ways. First, if a process is denied a further request while it has already held some resources, it must release all them and request them again together with the additional resources. Alternatively, if a process requests a resource that is currently held by another process, the operating system may pre-empt the resource from the second process and allocate it to the first one. Note that this approach is practical only when the resource of concern could be in some way restored to its original state.

2.11.4 Attacking Circular wait

In a systematic way, we may assign an index to each resource in the system, and all the processes are required to request for the resources in the order of increasing index. However this method may be inefficient. For example, a process may need to manipulate a resource with a larger index far before another resource associated with a smaller index is needed. To comply with this hold-and- wait prevention strategy, the second resource will have to be requested first, thus be held without utilization for a long time.

2.12 Deadlock avoidance

Different from deadlock prevention, where one of the four necessary conditions is prevented in some way, deadlock avoidance takes another approach, which is the progress of resource allocation in the operating system is monitored dynamically and whenever a deadlock is going to happen, some measure is taken to avoid it. Thus an evaluation process should be performed, when a resource allocation is requested, to make sure a deadlock will not happen.

Chapter 3- Requirements

The purpose of the chapter is to provide a list of requirements, functions and details for the process of constructing the system. These requirements set out what the software package should do, and define constraints on its operation and implementation.

Due to the incremental development approach followed for this project, the requirements have been through changes depending on the experience gained from the coding exploration. This helped to understand the feasibility of each requirement in terms of both design and implementation.

3.1 Original Requirements:

The name of the Topic is Concurrency and Visualization. This project then sub-divides it into the following subjects:

- Producer/Consumer
- Dining Philosophers

The problems and their corresponding solutions should be explained visually through the use of 2D graphics. If possible both problems should be in the same applet. The menu system should allow the user to choose the desired one.

Target Platform: UNIX and variants
POSIX standards

3.1.1 The Producer/Consumer Requirements:

- Give the user the ability to add
    - A Producer Process.
    - A Consumer Process
    - A Bounded Buffer.

The user of the program should be able to do the above with the use of a menu or by directly using the mouse. With the use of a mouse, by clicking on a certain rectangular region on the screen visible to the user the corresponding process or buffer should be displayed.

- Give the ability to the user to change the size of the buffer. (Maximum of: 10 slots and a minimum of: 1 slot) by using the menu system.
- Give the ability of the user to adjust the speed of the animation, through the use of a menu system.
- The program should highlight the important code on each process (Producer and Consumer) being executed.
- The program should allow the user to change the priorities of each of the processes even while the animation is executing.
- Three buttons should exist: Start, Stop and Reset.

- Start Button: Should activate the animation and start the processe(s) execution. Only if there is a Bounded Buffer.
- Stop Button: Should completely stop the animation. This should happen until all the locks are released. Processes should be terminated naturally.
- Reset Button: Should Initialize the Bounded Buffer. Only when the animation has stopped.

- Messages to the user:

  - When the producer is the only process active in the animation. When the buffer is filled completely, a message should be displayed informing the user that a deadlock has occurred and that the Consumer must be added for the deadlock to be removed.
  - When the Consumer is the only process active in the animation. When the buffer is completely empty, a message should be displayed informing the user that a deadlock has occurred and that the Producer should be added for the deadlock to be removed.
  - The animation cannot start unless a buffer exists. The start button should not be enabled. The message should state this.
  - Any other messages should be displayed that are important.

3.1.2 ***The Philosophers requirements:***

The user should have the ability to choose how many philosophers should be sitting on the table in the range of (Maximum of: 7 and Minimum of: 3).

- Two buttons should exist: Start, Stop.
  - Start Button: Should activate the animation and start the processe(s) execution. Provided there are Philosophers sitting around the table in the ranges stated above. If not Start should not be enabled.
  - Stop Button: Should completely stop the animation. This should happen until all the locks are released. Processes (Philosophers) should be terminated naturally. Stop and Start will be deactivated until the sequence is complete, by which then the Start button will be enabled.

- The program should highlight the important code on each process (Philosopher 1, Philosopher 2 …..) being executed.
- The program should allow the user to change the priorities of each of the processes even while the animation is executing.
- The program should indicate which Philosopher is:
  - THINKING
  - HUNGRY
  - EATING

3.1.3 ***Extra personal requirement:***

- To demonstrate knowledge of both Java and C by interfacing them together to produce the software package through some method and also to fulfil the above main requirement that of the POSIX standards.

3.2 ***Altered Original Requirements:***

These were the altered set of requirements.

- Platform: UNIX variants and Windows.
- Standards: Java standards (use of threads).
- Product: Producer/Consumer and Philosopher problems can be separate programs.
- Buttons: Reset Button excluded from Producer/Consumer program (Automated).

Chapter 4 – Methodology

The purpose of this chapter is to describe briefly several possible methodology principles, similar phases which exist between them and finally which of those described was chosen as opposed to the others for the development of this project's software according to the requirements outlined in Chapter 3.

4.1 Similarities in Methodologies

A software methodology is the set of rules and practices used to create computer programs. A heavyweight methodology has many rules, practices, and documents. It requires discipline and time to follow correctly. A lightweight methodology has only a few rules and practices or ones which are easy to follow. It allows adaptation to external forces, but depends on frequent testing such as regression testing and refactoring to ensure system quality.

When considering a software system, one can observe the system from a life cycle view. This means that the system is observed over time from the first notion of existence to the settlement of the system. Certain different software development life cycles have several notions in common and similar divisions of phases. These common and similar divisions can be grouped into the following [11]:

Analysis: To understand the activities that the software system is meant to support.
Design: To develop a detailed description of the software system.
Implementation: To formalize the design in an executable way.
Integration: To adjust the system to fit the existing software environment.
Test: To identify and eliminate the non desirable effects and errors and to verify the software system.

4.2 Risk Assessment

Oftentimes in projects, a plan is created for risk every time a system is developed, and an assessment is done of that risk. Both managers and technical personnel develop risk plans for current projects, based on historical data. However, quite often the risks are identified and then forgotten throughout the product's lifecycle. They are listed and put on the risk watch list, but not addressed until they become problems. To prevent this from happening, at each "project review" the Risk Owner should give the status of the mitigation or contingency plan, and the project team should decide to [11]:

1) Take action on the risk.
2) Eliminate the risk.
3) Retire the risk.

In order to handle this risk, new methods must be formulated essentially to be able to analyze and assess them. This is done with the use of software prototyping which can be used in the following three main areas:

- Exploration

- Experimentation
- Evolution.

Exploratory approaches use prototypes for finding requirements early. Rapid throwaway Prototyping and Boehm's spiral model fall within this classification. Experimental approaches use prototypes to investigate certain kinds of feasibilities or possibilities within the process of development. The feasibility investigated may be technical, increasing efficiency, etc. The evolutionary approaches use the philosophy of the prototyping process itself and employ it as the methodology for the software development process as a whole. Incremental development and evolutionary system development can both be viewed as types of evolutionary software development approaches.

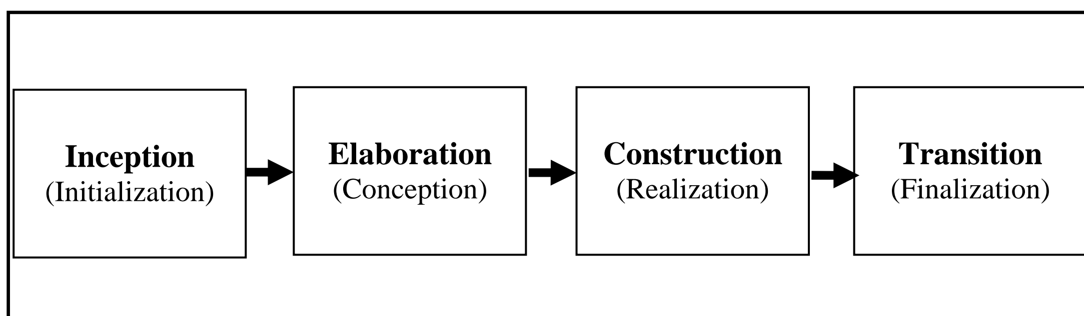


Figure 3 – Fowler's Incremental Software Development Process

(from Biel School of Engineering [7])

- **Inception:**
  The birth of a project and the initial idea.
  Establish the business rationale.
  Getting the commitment of the sponsors to go further.
- **Elaboration:**
  Collect more information.
  Create or get the user requirements.
  Establish the core architecture of the software and do the planning.
- **Construction:**
  Produce the software iteratively and each iteration yielding a workable piece of software.
- **Transition:**
  Do the beta testing;
  Make the performance optimization.

Prototyping is the use the development of one or two working versions of various aspects of a system. It is not production code but it may eventually become pre-production code or it may be completely discarded. In the prototyping effort, maintainability of the code is not the main concern nor is documenting it. Code resulting from prototyping is often used to train the programmers. Only after it has written specifications resulting from the experience with the prototype should the programmer or team start the formal development process. A prototype produces "running" software and the production development produces "working" software.

4.3 Implemented Methodology

In the case of this project the reason for using incremental development as opposed to the waterfall method was mostly due to the unfamiliar topic area. This was a risk that needed to be seriously taken into account. On one hand, in the waterfall method each phase is formally ended with fully elaborated written documentation. This is time-consuming to produce, not suitable for this project. User requirements are *frozen* after requirements analysis and therefore cannot be changed during development process. In the real world the customer can only assess the project at the very end, when the finished software is handed over; at that time major mistakes may be uncovered. The waterfall model is a linear approach, quite inflexible. At each phase, feedback to previous phases is possible (but is discouraged in practice). On the other hand incremental development allows a project to construct the software in incremental stages where each stage adds additional functionality. Each stage consists of design, code, unit test, integration test and delivery. It allows the project team to put functional software into the hands of the customer much earlier than the waterfall model. Stages can be planned in such a way that can determine what functionality needs to be done first i.e. choosing to deliver the most important functionality to the customer first. It can provide tangible measures of progress but requires careful planning at both the project management level and the technical level. The project management team monitors the progress of the project by examining the end product and duration of each increment, building an increasingly accurate picture of the size of the project and the rate of progress. Although the Incremental development can be viewed as a type of evolutionary software approach, in the case of this project the author also used it as an exploratory and experimental approach.

The next part of the dissertation shows emphasis on describing the design, implementation and testing of the Producer/Consumer problem. The internal software for the Dining Philosophers problem was also developed (The code is available on the accompanying CD). Visualisation software for that problem was not developed, but the concept is very similar to that of the producer/consumer.

Chapter 5 – Design

The purpose of this chapter is to describe the main design of the UI of the Producer/Consumer Software and the various implementation approaches that could be used to accomplish it, these being a proposed design aimed at a Java and C implementation and two further designs aimed at 100% Java implementation. With each implementation approach its corresponding problems and risks are described.

5.1 Design of Producer/Consumer Visualisation

Detailed desired designs were created for the Producer/Consumer problem given below, which the author strictly tried to abide by them.
Design includes:

- Menu System
    - Edit
    - Options
    - Help
- Animation System
- Control Panel
    - Producer Priority
    - Consumer Priority
    - Producer Code Steps
    - Consumer Code Steps

The preferred final Interface is shown below in the diagram. The menu system is positioned on the top part of the screen. Below it is the buttons panel and to the right is another panel which include the priorities of each process and the major code listing.

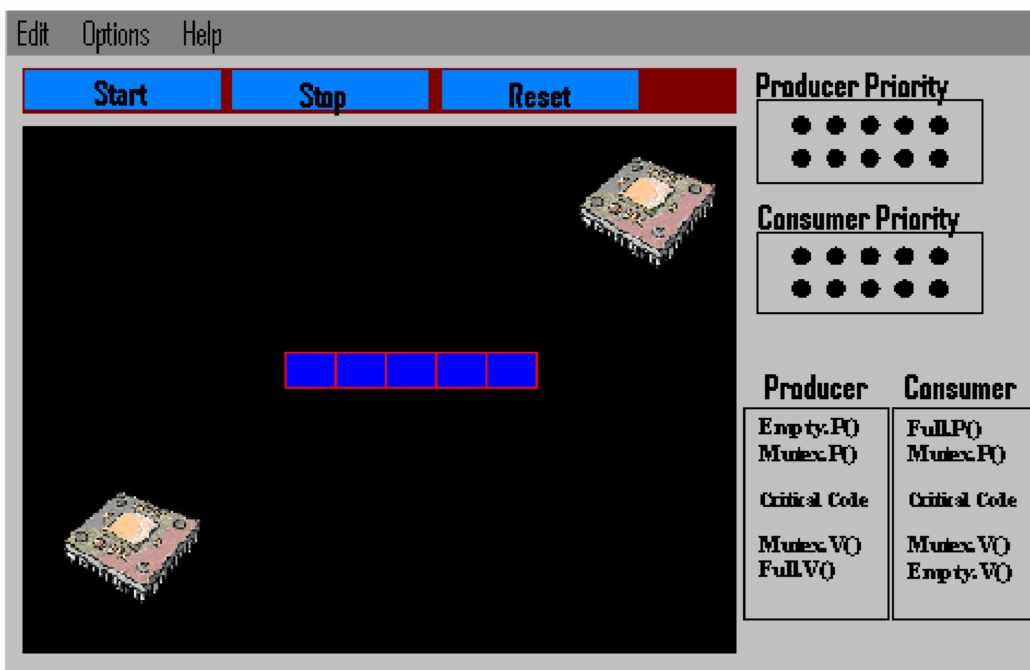


Figure 4 – The Producer/Consumer GUI look

The GUI is divided into 5 parts:

- Menu System.

Figure 5 – The Main Menu Component

- Buttons Panel

Figure 6 – The Main Buttons Component

- Animation Panel

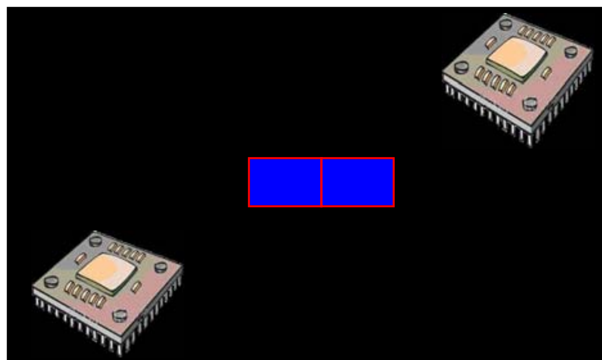


Figure 7 – The Main Animation Component

- Control Panel

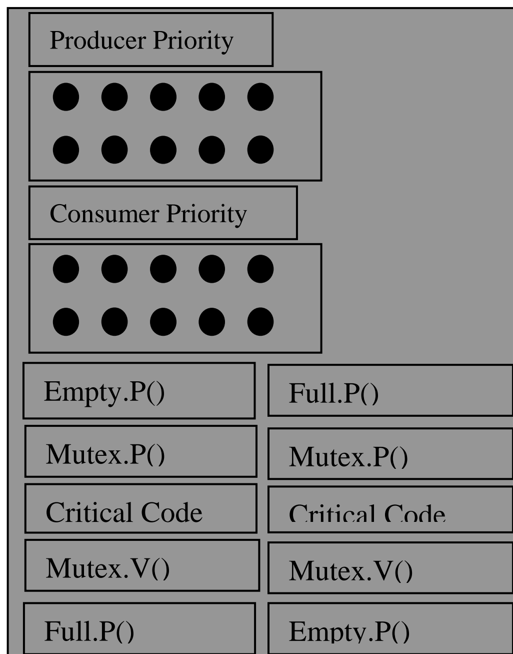


Figure 8 – The Control Panel Component

The next figure tries to show the animation path of the Item created by the producer targeted for the next available slot in the Memory Bounded Buffer and the path taken when its turn comes for it to be retrieved by the Consumer. A formula close to that of **y=x^3** was used to simulate the below movement. The animation sequence can be divided into four parts.

- Item leaving Producer heading for the Buffer.
- Item inside Buffer moving next to the End of the Buffer or next to the Previous inserted Item
- Item leaving Buffer heading toward Consumer.
- All Items inside buffer shifting one place to the right.

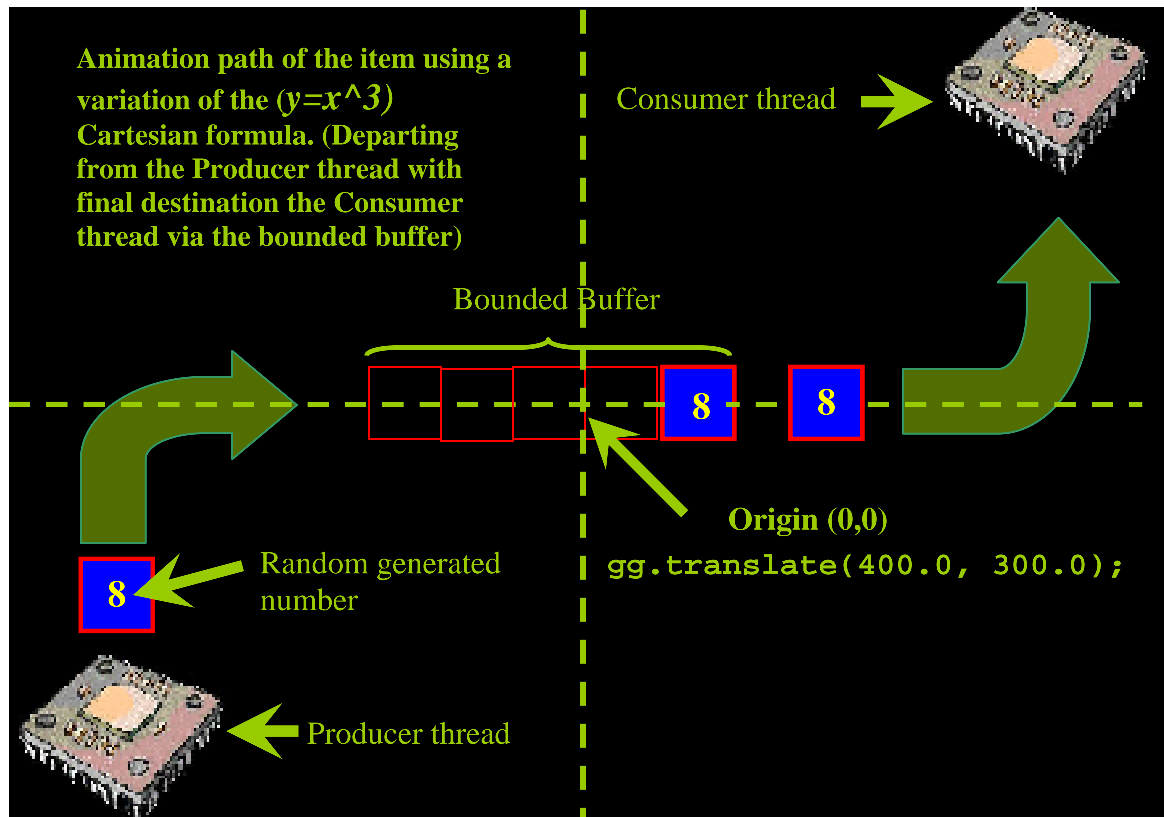


Figure 9 – The Animation Path of the Item.

The final applet went through a variety of designs.

5.2 *Java & C implementation using JNI*

The first design was to include both the Java and C language with the use of JNI (Java Native Interface). JNI defines a standard naming and calling convention so the Java virtual machine can locate and invoke native methods. JNI is built into the Java virtual machine so the Java virtual machine can invoke local system calls to perform input and output, graphics, networking, and threading operations on the host operating system.

A few problems arose during the implementation. On making a simple graphical user interface which reflected somewhat that of the final product the communication of the java class would not work correctly when calling the C native function under Solaris 2 platform, although it worked fine in Windows. The problem specifically was that the function being called would not execute until the GUI interface was terminated either by closing the window or pressing Ctrl – C. The design was terminated as soon as this problem arose. At the time, possible reasons for this problem where researched. The main reason that seemed plausible after some time was that:

- Solaris 2 handles threads slightly differently than Windows. On Windows, a Java SWING program has its own daemon thread for the GUI. Therefore calling the C DLL from Java program would run on the program thread in parallel with the daemon swing thread. On Solaris this is not so, the GUI needs to be declared on a separate thread by the programmer.

A sample of the Design although not complete is given in the diagram on the next page.

All the system calls would be implemented in the C language and then compiled in SO in UNIX equivalent to a DLL in Windows. As soon as the Start Button was pressed in the JAVA ApplicationGUI it would call a method in the compiled SO listed in the ApplicationGUI.h header file.

The DLL or SO would

- Create a shared memory.
- Create two Child Processes
    - Producer
    - Consumer
- Create the Semaphores
    - Empty
    - Full
    - Mutex

} Using POSIX standards

The sample C code is demonstrated in the next pages.

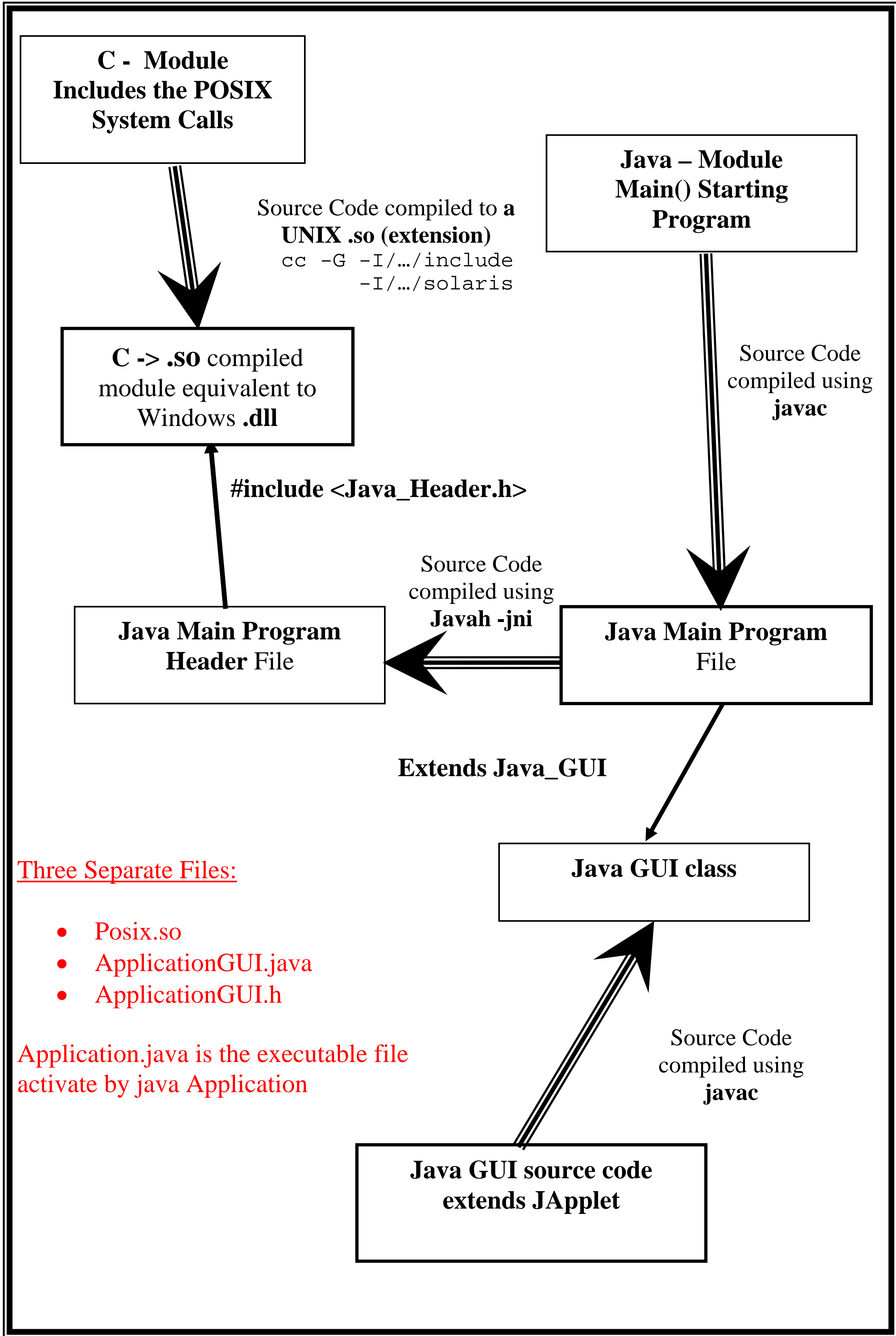
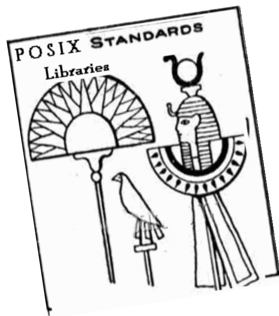


Figure 10 – Java and C Interfacing using JNI

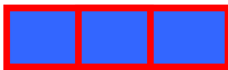



```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <unistd.h>
#include <sys/sem.h>
#include <signal.h>
#include <time.h>
```

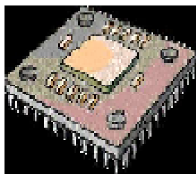
Declarations Definitions

```
// Variables for Shared Memory.
  int    ShmID;
  int    *buffer;
  pid_t  childpid1, childpid2;

  // Variables for Semaphores.
  int    mutexid;
  int    fullid;
  int    emptyid;

void semaphore_p (int);
void semaphore_v (int);
void clear_all (int);
```

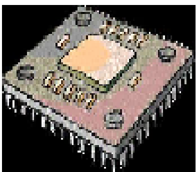
Memory Shared Buffer



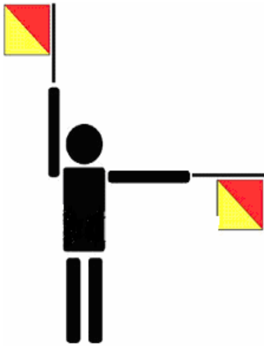
```
ShmID = shmget(IPC_PRIVATE, BUFFSIZE*sizeof(int),
IPC_CREAT | 0666);
                  if (ShmID < 0) {
                  }

 buffer = (int *) shmat(ShmID, NULL, 0);
               if ((int) buffer == -1) {
               }
```



```
Childpid1=fork();

  if (childpid1==-1) {
  }
  if (childpid1==0)  {
    // Code placed here for Child Process 1
      exit(0);
  }
```



```
Childpid2=fork();

  if (childpid2==-1) {
  }
  if (childpid2==0)  {
    // Code placed here for Child Process 1
      exit(0);
  }
```



```
// Define three Semaphores. Type (int)
 Mutexid = semget(IPC_PRIVATE, 1, IPC_CREAT |  0666);
 emptyid = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666);
 fullid      = semget(IPC_PRIVATE, 1, IPC_CREAT | 0666);

if( semctl(mutexid, 0, SETVAL, 1) == -1)
{// Check for initialization error}
if( semctl(emptyid, 0, SETVAL, buffer) == -1)
{// Check for initialization error}
if( semctl(fullid, 0, SETVAL, 0) == -1)
{// Check for initialization error}

void semaphore_p(int semid)
{
   // initialize values
   struct sembuf op;
   op.sem_num =  0;
   op.sem_op  = -1;        // dec by one
   op.sem_flg =  0;        // no flags

   if( semop(semid, &op, 1) == -1)
      printf("semaphore_p: semop error\n");
}

void semaphore_v(int semid)
{
   // initialize values
   struct sembuf op;
   op.sem_num = 0;
   op.sem_op  = 1;        // inc by one
   op.sem_flg = 0;        // no flags

   if( semop(semid, &op, 1) == -1)
      printf("semaphore_v: semop error\n");
}
```

Code Sample 1 – C POSIX Standard system calls

5.3 100% Java Implementation

It was later decided to go on to another design, which seemed more reliable, using 100% Java in the design and implementation. The reason for this was that Java allows for the ultimate degree of code portability. Finally the same program can be executed securely on any computer, over any operating system, as long as the Java Virtual Machine runs on the underlying system. Through the use of its extensive class libraries, programmers have discovered that development time and effort are much reduced: the language provides extensive resources for development of user interfaces

5.3.1 A Java Single Applet Design

With this approach many other problems were experienced. Through the use JApplet there were times in the implementation that the Applet would not work in any Browser but would

work in appletViewer. One major problem was the addition of a menu system within the same applet. When run on a browser the menu system could not be activated while the animation was running at the same time. When it did activate, due to continued clicks by the user the response was very slow, this problem did not exist on the appletViewer. The main reason that seemed plausible after some time was that:

- within the Browser exists a security manager that checks all sensitive operations, in our case defined in the Producer/Consumer applet program UI. These operations violated certain rules and therefore were disabled by the interpreter at times.

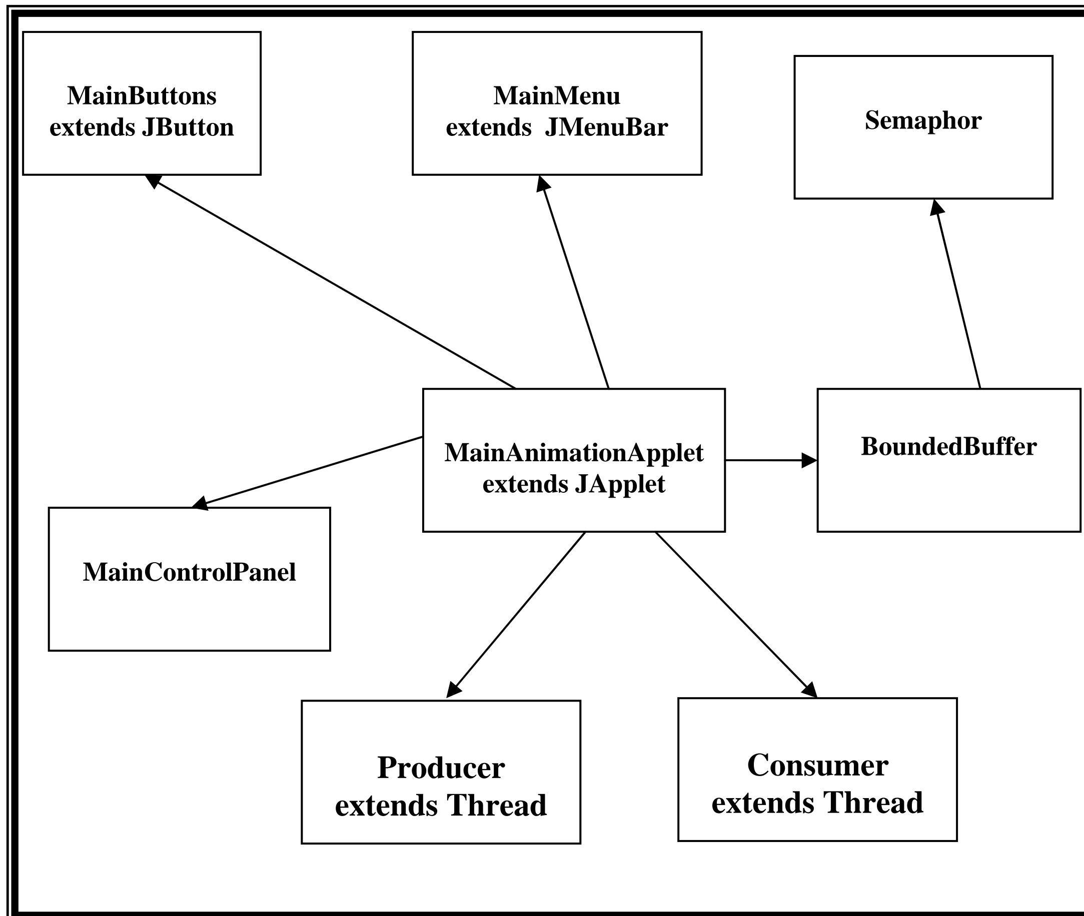


Figure 11 – 100% Java class implementation

Above is a diagram depicting the various classes that would be used in the 100% Java implementation.

5.3.2 *A Three Applet Java Design*

It was decided to create a menu on a different applet, and have these two applets communicate with each other. This worked fine, until a third one was needed to act as the control panel which would add some direct functionality for the user. A new problem arose when the main applet had to communicate with the control panel. Under all the browsers including Windows Explorer, Netscape and UNIX Solaris Mozilla the applets could not communicate.

The probable cause after researching on the web seemed to be that browsers add certain restriction on applet communications. This problem could not be bypassed and therefore, an attempt to re-engineer the original single applet design was attempted. However, when it comes to Swing and JApplets, things are a little more complicated. The mistake made was that the author was drawing directly on the JApplet canvas or on other top-level Swing component. This was fatal mistake, instead a separate component should have been used as a drawing surface, and this should have been added to the JApplet. It was necessary to write a class to represent the drawing surface. Programming a JApplet that does custom drawing always involves writing at least two classes: a class for the applet itself and a class for the drawing surface. Typically, the class for the drawing surface is defined as a subclass of `javax.swing.JPanel`, which by default is nothing but a blank area on the screen. A JPanel, like any JComponent, draws its content in the method paintComponent. In the case of the program being built, the drawing surface was made as an internal class within the JApplet itself.

Below is the diagram depicting the three applet design. The communication is done through the use of the statement getAppletContext() which is imported from java.applet.

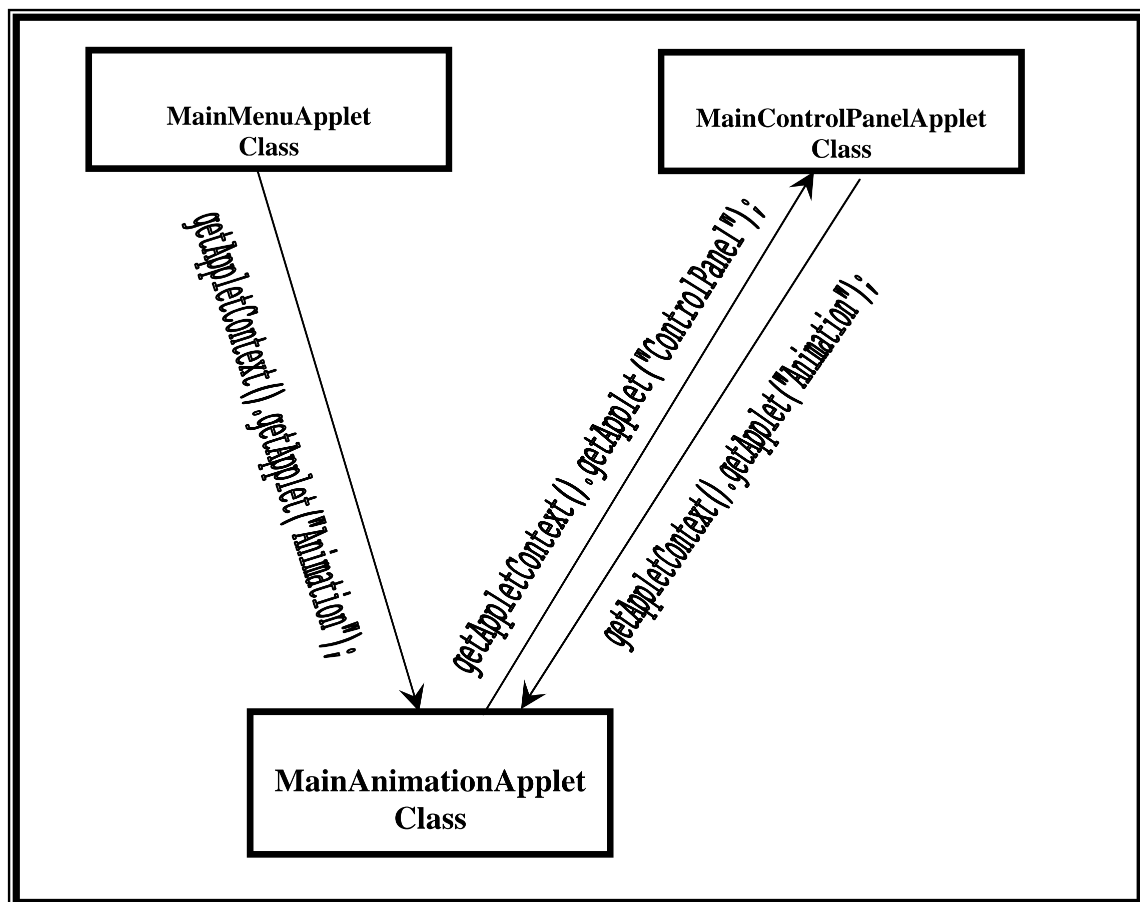


Figure 12 – The three JApplet Mechanism

On the next page are the set of **Class diagrams** with their respective variables and methods.

5.4 Producer/Consumer Class Diagrams

MainAnimationApplet
- mainButtons:MainButtons; - mainMenu:MainMenu; - controlPanel:ControlPanel; - drawingSurface:Display; - timer:Timer; - timerPerformer:ActionListener; - buf:Image; - buf2D:Graphics2D; - theBuffer:BoundedBuffer; - theProducer:Producer; - theConsumer:Consumer; - messagetimerCount:int=0; - theItemCoordinateX:double=-100; - theItemCoordinateY:double=-100; - theItemCoordinateQ:double=10; - theItemCoordinateW:double=100; - theCorrectAnimationSequence:int=0; - theCorrectItemNo:int=0; - theRequestedSpeed:double=2; - theRequestedSizeOfTheBuffer:int=3; - memItemPosX:double=312; - theProducerMustWAIT:boolean=false; - theConsumerMustWAIT:boolean=false; - memItemPosY:int=169; - howManyItemsAreStored:int=0; - moveToTheNextEmptySlot:int=0; - displayTheProducerThread:boolean=false; - displayTheConsumerThread:boolean=false; - displayTheBoundedBufferSlots:boolean=false; - theTracker:MediaTracker = null; - theProducerConsumerImage:Image; - whichMessageToDisplay:int=0; - theAnimationMustTerminateAsSoonAsPossible:boolean=false; - theAnimationHasBegun:boolean=false; - hasTheProducerAlreadyBeenStarted:boolean=false; - hasTheConsumerAlreadyBeenStarted:boolean=false; - highlightTheProducerArea:boolean=false; - highlightTheConsumerArea:boolean=false; - theProducerPriority:int = 5; - theConsumerPriority:int = 5; **(MainAnimationApplet continued on next page)**

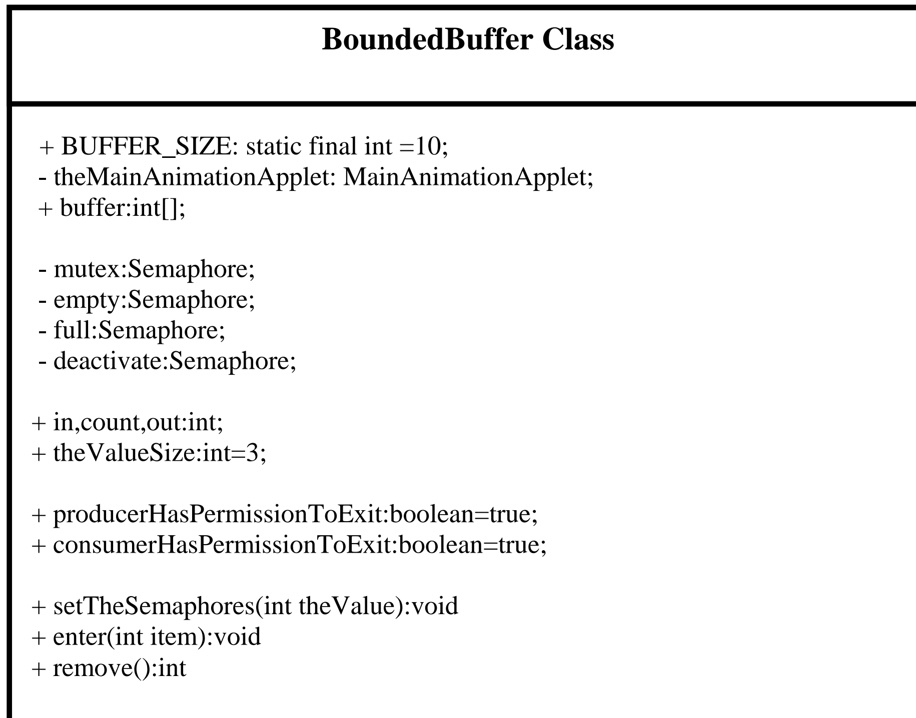
+ class Display extends JPanel

- + paintComponent(Graphics g): void
- + update(Graphics g):void
- + initTheInvisibleGraphicsContext(Graphics g):void
- + displayTheBoundedBuffer(Graphics g):void
- + whichAnimationSequence(Graphics g):void
- + displayTheProducerThreadImage(Graphics g):void
- + displayTheConsumerThreadImage(Graphics g):void
- + noOfItemsInTheBuffer(Graphics g):void
- + theVariousApplicationMessages(Graphics g):void

+ synchronized getTheThreadsCodeLocationAndDisplayIt(int threadCodeLocation
,int whichThread):void
+ synchronized getTheThreadsCodeLocationAndAnimateIt(
int CorrectAnimationSequence,int CorrectItemNo):void
+ getTheUsersMainMenuChoice(int aChoice, int someExtraInfo):void
+ getTheUsersControlPanelChoice(int aProducerPriority
,int aConsumerPriority):void
+ getTheUsersButtonChoice(int aButton):void
+ stopTheConcurrencyandAnimation():void
+ startTheConcurrencyandAnimation():void
- class MouseHandler extends WindowAdapter implements MouseMotionListener,
MouseListener

- +synchronized mouseMoved(MouseEvent evt):void
- +synchronized mousePressed(MouseEvent evt):void

Class Diagram 1 – MainAnimationApplet.java


BoundedBuffer Class

+ BUFFER_SIZE: static final int =10;
- theMainAnimationApplet: MainAnimationApplet;
+ buffer:int[];

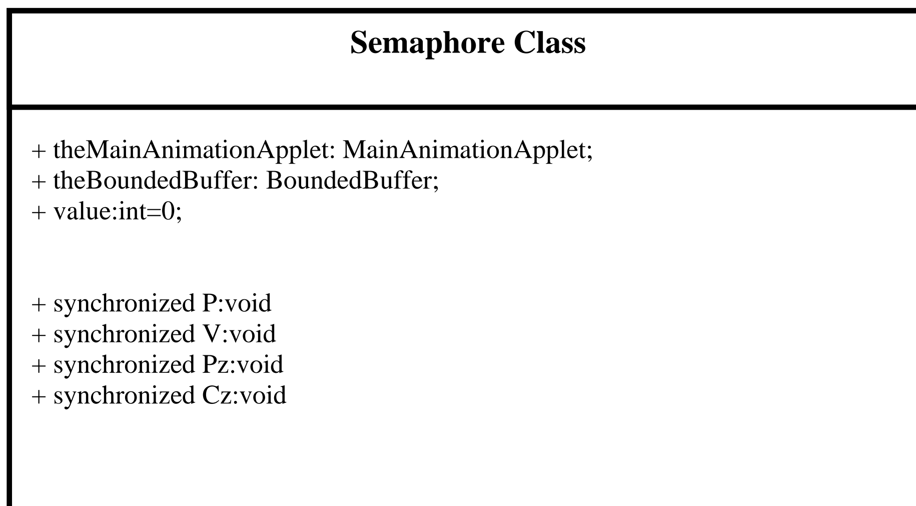
- mutex:Semaphore;
- empty:Semaphore;
- full:Semaphore;
- deactivate:Semaphore;

+ in,count,out:int;
+ theValueSize:int=3;

+ producerHasPermissionToExit:boolean=true;
+ consumerHasPermissionToExit:boolean=true;

+ setTheSemaphores(int theValue):void
+ enter(int item):void
+ remove():int

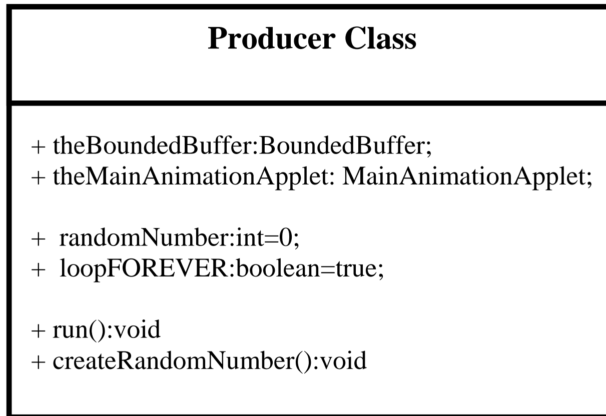
Class Diagram 2 – BoundedBuffer.java


Semaphore Class

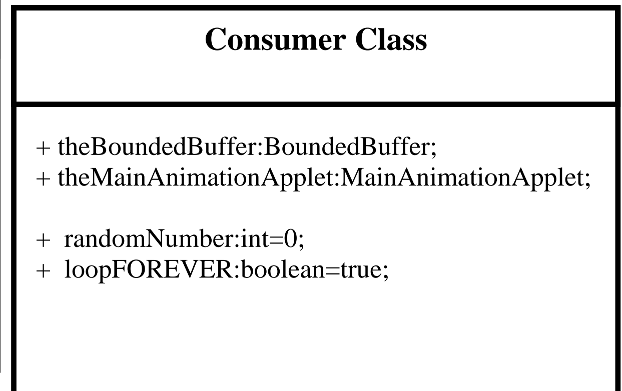
+ theMainAnimationApplet: MainAnimationApplet;
+ theBoundedBuffer: BoundedBuffer;
+ value:int=0;

+ synchronized P:void
+ synchronized V:void
+ synchronized Pz:void
+ synchronized Cz:void

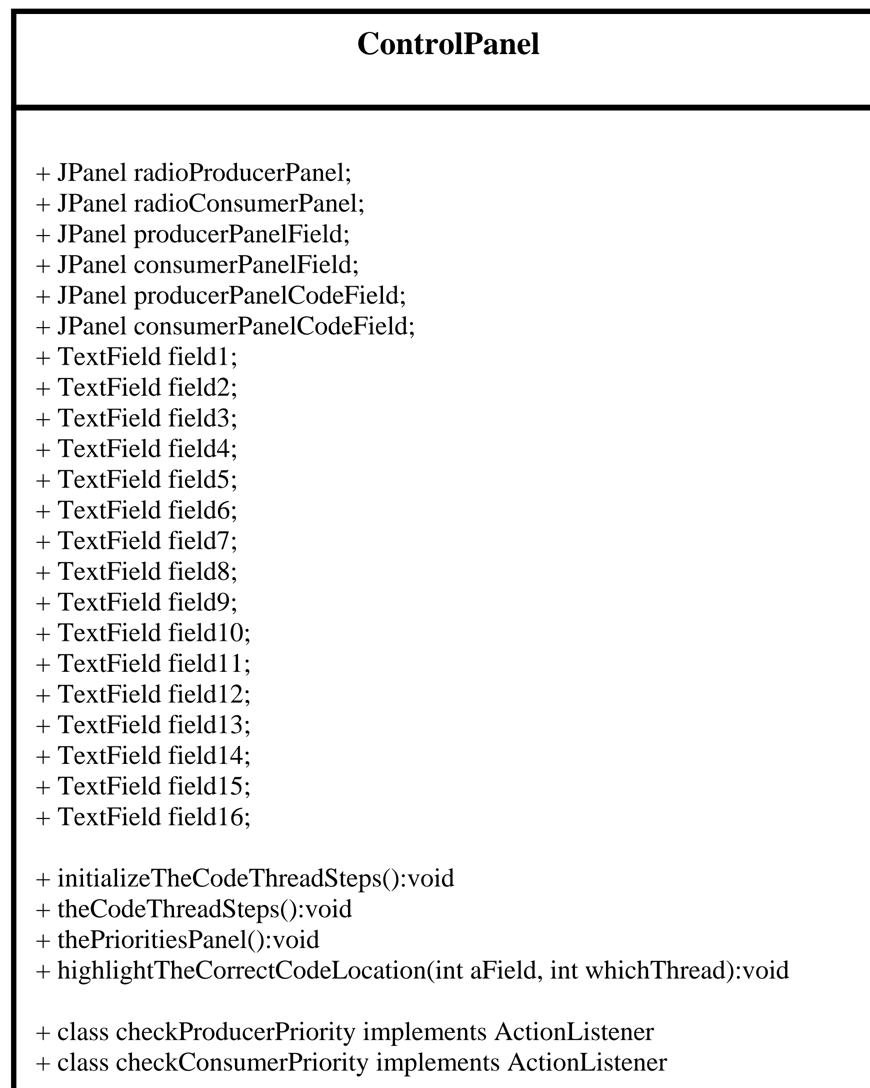
Class Diagram 3 – Semaphore.java



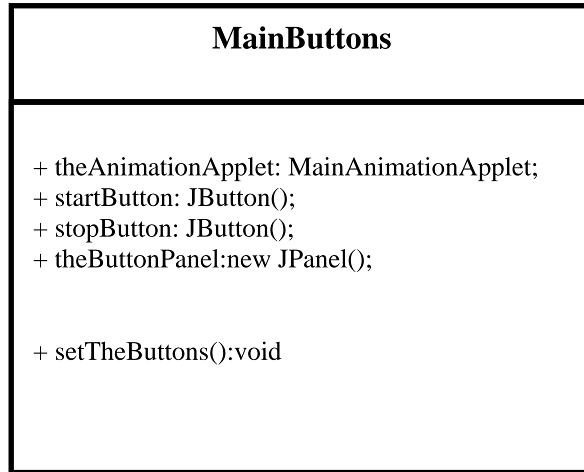
Class Diagram 4 – Producer.java



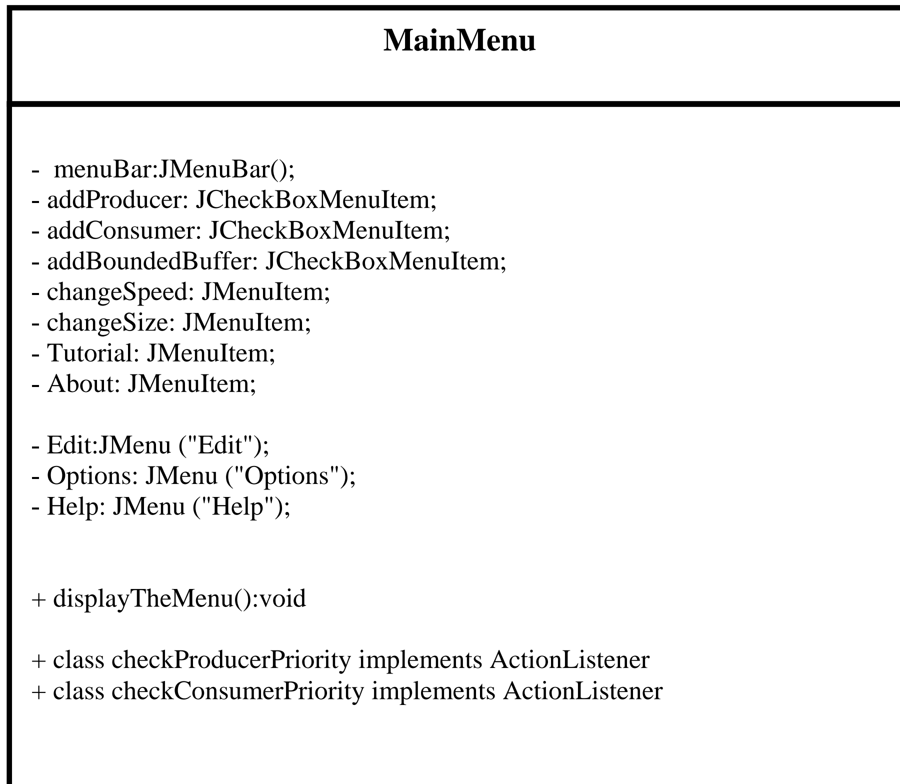
Class Diagram 5 – Consumer.java



Class Diagram 6 – ControlPanel.java



Class Diagram 7 – MainButtons.java



Class Diagram 8 – MainMenu.java

Chapter 6 – Implementation

The purpose of this chapter is to describe the implementation of the design developed in the previously and also explain certain interesting algorithms used to accomplish the requirements specified in Chapter 3.

6.1 The Basic Algorithm

The basic formula for the Producer Consumer problem is annotated in the diagram below.

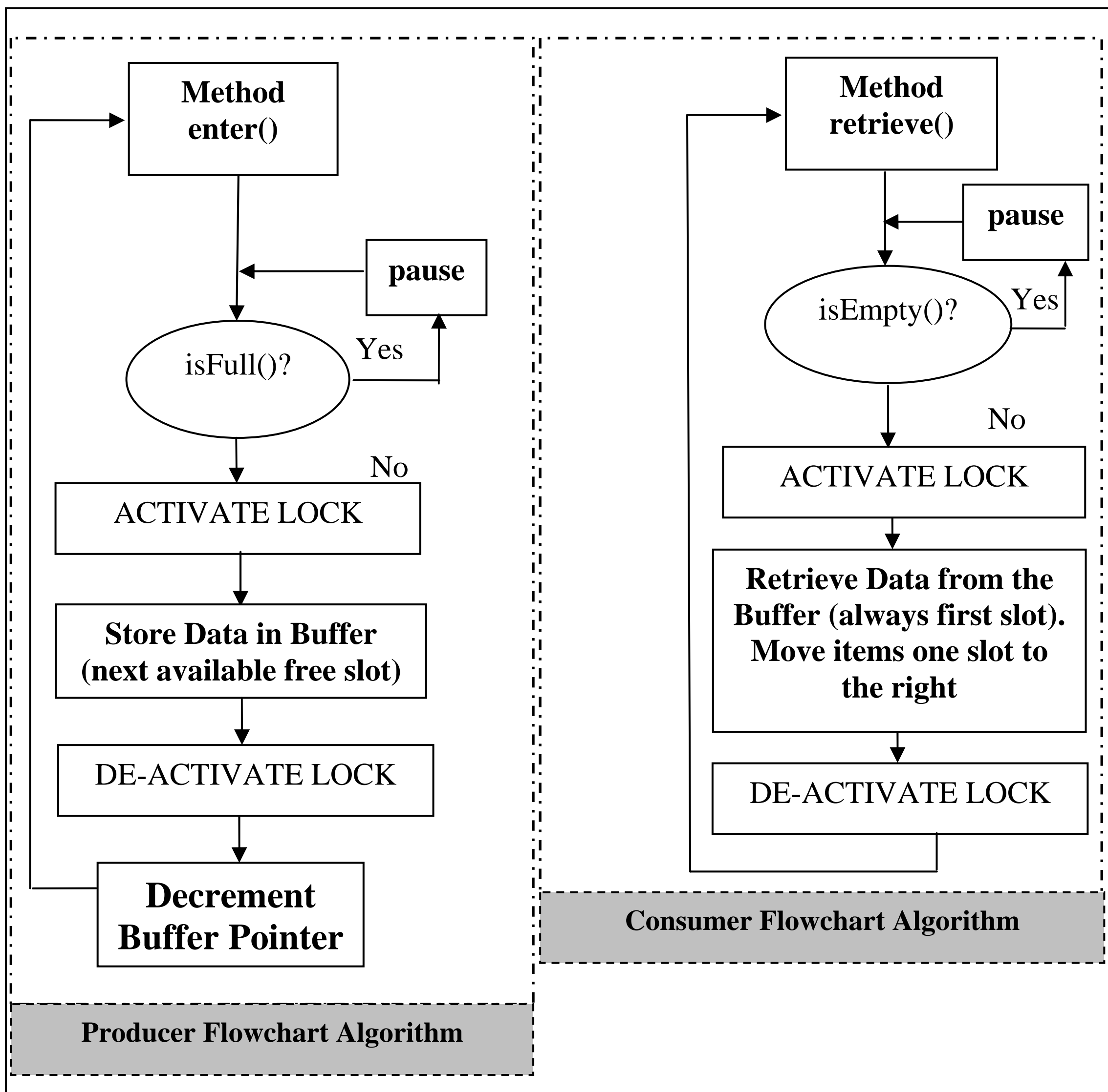


Figure 13 – Producer/Consumer Concept

Although not depicted in the diagram above but will be mentioned further on in this chapter the Producer and Consumer Threads interact with the main class the MainAnimationApplet at certain intervals for an animation sequence. The animation is called from within the critical

regions of each thread. When the producer inserts items into the Bounded Buffer, the thread sleeps for a while, until the animation of the item being inserted into the buffer visually is finished, then it awakens and continues from were it left of, likewise when the consumer retrieves an item from the Bounded Buffer the thread sleeps until the items being removed is visually completed and awakens continuing from were it left of.

6.1.1 ***The Bounded Buffer:***

The Bounded Buffer is created in the BoundedBuffer Class of an array that holds a maximum of ten slots. The user though can adjust the size used by the Producer and Consumer thread by adjusting the value from the Control Panel on the right of the screen. The Bounded Buffer can have either maximum of ten or a minimum of one slot for the animation to work. Even though the size can change by the user the declared size is still ten elements. But with the use of two Semaphores, these being full and empty declared in the BoundedBuffer class the elusion is created that the Buffer has changed size. The default size set by the program by initialization is of size five.

6.1.2 ***The Producer:***

```
empty.P();
mutex.P();

// Critical Code //

mutex.V();
full.V();
```

The first line of code empty.P() checks to see if the Buffer is Full. The statement calls the method P() in the Semaphore class. This method will check if the value of the variable Value defined in that class is Zero. IF the value is zero it means that the buffer is full and that the Producer should wait for a while until its counterpart the Consumer retrieves at least one item. If the value is not zero then it is decremented by one and the flow of control returns back to the next statement in the algorithm. This second statement mutex.P() calls the same method in the Semaphore class but under a different instance. This is a binary declaration called a mutex, which can either have a value one or zero. When the value is one it means that the Producer can have access to its Critical Section and therefore can continue to the next line of code the critical part of the method. The critical section inserts a random value which has been generated in the Producer thread and stores it into the shared Buffer array declared in the Bounded Buffer Class. When this has been done the critical section of code also calls a method in the Animation Class to create the visualization of the task on the screen. This visualization includes an item being moved from the Producer image to the graphical Buffer situated in the centre. In that method the Producer thread sleeps for a while the animation is executing and awakes when it is finished. Returning back to the enter method in the Bounded Buffer class the next piece of code increments the value of the variable Value in the Semaphore class, thereby releasing the lock and then notifying the other thread, the Consumer that it is permitted to proceed to its own Critical Section.

6.1.3 The Consumer:

```
full.P();
mutex.P();

// Critical Code //

mutex.V();
empty.V();
```

On the other hand the consumer's code is slightly different in its sequence. The first line of code checks to see if the Bounded Buffer is empty, by calling the method P() in the Semaphore class. IF the value of the variable Value is zero, it means that the Bounded Buffer is empty and therefore the Consumer must wait until the Producer enters at least one item for the Consumer to continue to the next line of code. If the value of Value is not zero, the consumer decrements that value and return back to the retrieve method defined in the Bounded Buffer class, whereby it executes the mutex() line. As mentioned in the Producer section, the mutex acts as a Binary switch. When equal to zero the consumer must wait until the Producer releases the lock. If not equal to zero the consumer is than free to proceed to the Critical Section which is to retrieve the value at location zero in the buffer, since the Data Structure is FIFO (the first item to enter is the first to be removed), then to call the method in the Animation class simulate the visualization of the item being removed from the buffer and entering the Consumer class. As with the Producer, the Consumer thread as well sleeps for some time while the animation is executing and awakes when it is finished. After this has been achieved the Consumer must release the lock by executing the mutex.V() which will increment the Value from zero to one and thereby allowing the Producer to proceed if it is its turn by the scheduler to execute its Critical Section of code. The next line of code empty.P() increments the value by one, to indicate that the buffer is
now one less item.

One of the requirements listed in chapter 3 stated that the package should highlight graphically the important steps of code being executed by each process. This graphical representation as stated in the design in chapter 5 would be placed on the right side of the JApplet in the area of the Control Panel.

To accomplish this task extra code had to be added in the Bounded Buffer Class in the methods:

- Public void enter(int Item)
- Public int retrieve()

to allow for this Real Time Code display. The extra code is depicted on the diagrams in the next two pages in BOLD format to distinguish it from the previous code which has been already explained in previous pages. To accomplish this, all the code that involved calling the Main Animation applet and any other conditional statements had to be written in between locking mechanisms to stop the other process in this case the consumer from intervening. Five steps had to be shown for each process. In the case of the producer:

```
empty.P()
mutex.P()
CRITICAL SECTION
```

Mutex.V()
Full.V()

Therefore an extra two Locking and Unlocking mechanisms had to be inserted.

6.2 Amendments to the Producer Algorithm:

Mutex.P() // Lock
// communicate with the Animation Applet and highlight the empty.P() code
// Let the producer thread sleep for a while.

Empty.P()
Mutex.P()
Critical Code
Mutex.V()
full.V()

Mutex.V() //Unlock

Empty.P() // execute the Semaphore

Mutex.P() // Lock
// communicate with the Animation Applet and highlight the mutex.P() code
// Let the producer thread sleep for a while.

Empty.P()
Mutex.P()
Critical Code
Mutex.V()
full.V()

Mutex.V() // Unlock

Mutex.P() // Lock

//communicate with the Animation Applet and highlight the CRITICAL SECTION code

Empty.P()
Mutex.P()
Critical Code
Mutex.V()
full.V()

// Let the producer thread sleep until the animated item leaving the consumer reaches the
// bounded buffer, enter the value of the item in the array
// communicate with the Animation Applet and highlight the mutex.V() code

Empty.P()
Mutex.P()
Critical Code
Mutex.V()
full.V()

Mutex.V() // Unlock

Mutex.P() // Lock
// communicate with the Animation Applet and highlight the full.P() code
// Let the producer thread sleep for a while.
Mutex.V() // Unlock

Empty.P()
Mutex.P()
Critical Code
Mutex.V()
full.V()

Full.P() // execute the Semaphore

The next page displays the diagrammatic illustration presented in terms of code. The added code is in Bold.

```
public void enter(int item)

 // First Section of Thread
       //*****************************************************************
       mutex.P();
       if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
       {
          theMainAnimationApplet.getTheThreadsCodeLocationAndDisplayIt(3,0);
          if (in==theValueSize)
          {
             if (theMainAnimationApplet.displayTheConsumerThread==false)
             {
                //Producer says DEADLOCK!!!!");
                theMainAnimationApplet.whichMessageToDisplay=2;
                theMainAnimationApplet.repaint();
             }
          }
       }
       mutex.V();

       empty.P();
        theMainAnimationApplet.whichMessageToDisplay=0;

       // Second Section of Thread
       mutex.P();
       if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
       {
          theMainAnimationApplet.getTheThreadsCodeLocationAndDisplayIt(4,0);
       }
       mutex.V();

       // Third Section of Thread: CRITICAL SECTION
       mutex.P();
        if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
       {
          theMainAnimationApplet.getTheThreadsCodeLocationAndDisplayIt(5,0);
          buffer[in]=item;
          in++;
          theMainAnimationApplet.getTheThreadsCodeLocationAndAnimateIt(1,item);
          theMainAnimationApplet.getTheThreadsCodeLocationAndDisplayIt(6,0);
       }
       mutex.V();

       // Fourth Section of Thread
         mutex.P();
         if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
         {
            theProducerSection=0;
            theMainAnimationApplet.getTheThreadsCodeLocationAndDisplayIt(7,0);
         }
         mutex.V();
         full.V();
}
```

Code Sample 2 – Amended Producer Code

Java uses the synchronized keyword to indicate that only one thread at a time can be executing in this or any other synchronized method of the object representing the monitor. The Java language and runtime system support thread synchronization through the use of monitors. In general, a monitor is associated with a specific data item (a condition variable) and functions as a lock on that data. When a thread holds the monitor for some data item, other threads are locked out and cannot inspect or modify the data. The code segments within a program that access the same data from within separate, concurrent threads are known as critical sections. In the Java language, critical sections are marked with the synchronized keyword. Generally, critical sections in Java programs are methods. Smaller code segments can be marked as synchronized also, but this violates object-oriented paradigms and leads to confusing code that is difficult to debug and maintain. It is best to use synchronized only at the method level. The Java runtime system allows a thread to re-acquire a monitor that it already holds because Java monitors are re-entrant. Re-entrant monitors are important because they eliminate the possibility of a single thread deadlocking itself on a monitor that it already holds.

Consider the class in the diagram below.

```
class Reentrant
{
   public synchronized void method1()
   {
            method2();
            System.out.println("This is method1()");
   }

   public synchronized void method2()
   {
            System.out.println("This is method2()");
   }
}
```

Code Sample 3 – Synchronization in Java

Re-entrant contains two synchronized methods: method1() and method2(). The first synchronized method, method1(), calls the other synchronized method, method2().

When control enters method method1 the current thread acquires the monitor for the Re-entrant object, method1() calls method2() and because method2() is also synchronized the thread attempts to acquire the same monitor again. Because Java supports re-entrant monitors, this works correctly. The current thread can acquire the Re-entrant object's monitor again and both method1() and method2() execute correctly.

A thread can call wait() to block and leave the monitor until a notify() or notifyAll() places the thread back in the ready queue to resume execution inside the monitor when scheduled. A thread that has been sent a signal is not guaranteed to be the next thread executing inside the monitor compared to one that is blocked on a call to one of the monitor's synchronized methods. Also, it is not guaranteed that the thread that has been waiting the longest is the one woken up with a notify(); an arbitrary thead is chosen by the JVM. When a notifyAll() is

called to move all waiting threads back into the ready queue, the first thread to get back into the monitor is not necessarily the one that has been waiting the longest. Therefore Java monitors are technically called signal-and-continue.

Below is a detailed analysis of what exactly each statement does when it is executed by the Java JVM [15].

Wait

A wait invocation results in the following actions:

- If the current thread has been interrupted, then the method exits immediately, throwing an InterruptedException. Otherwise, the current thread is blocked.
- The JVM places the thread in the internal and otherwise inaccessible wait set associated with the target object.
- The synchronization lock for the target object is released, but all other locks held by the thread are retained. A full release is obtained even if the lock is re-entrantly held due to nested synchronized calls on the target object. Upon later resumption, the lock status is fully restored.

Notify

A notify invocation results in the following actions:

- If one exists, an arbitrarily chosen thread, for example the Producer, is removed by the JVM from the internal wait set associated with the target object. There is no guarantee about which waiting thread will be selected when the wait set contains more than one thread. The Producer must re-obtain the synchronization lock for the target object, which will always cause it to block at least until the thread calling notify releases the lock. It will continue to block if some other thread obtains the lock first.
- The Producer is then resumed from the point of its wait.

NotifyAll

A notifyAll works in the same way as notify except the steps for all threads in the wait set for the object. However, because they must acquire the lock, threads continue one at a time.

Interrupt

If Thread.interrupt is invoked for a thread suspended in a wait, the same notify mechanics apply, except that after re-acquiring the lock, the method throws an InterruptedException and the thread's interruption status is set to false. If an interrupt and a notify occur at about the same time, there is no guarantee about which action has precedence, so either result is possible.

The sequence of the animation depends on the variable theCorrectAnimationSequence defined in the MainAnimationApplet class which is equated in the method getTheThreadsCodeLocationAndAnimateIt() with the value passed in by enter() method in the BoundedBuffer Class in the critical section.

When the Producer is executing, the value of the theCorrectAnimationSequence can be either 1 or 2. The value initially is 1, this means the Item is from the Producer and heading toward the Bounded Buffer. At this point the Item is moving in the form of the formula y=x^3. In the region of values (-100<X<=0) (-100<Y<0). When it reaches the end of the buffer the value changes to 2 and the movement of the Item moves horizontally for values in the range -100>X<100 while y remains constant. This range adjusts accordingly depending on how many items are stored in the memory slots. Each time an item is inserted the range decreases on the right side. The table below depicts the range of values when the buffer slots get filled.

Item	Range	Difference
First	-100<X<100	0
Second	-100<X<80	20
Third	-100<X<60	40
Fourth	-100<X<40	60
Fifth	-100<X<20	80
Sixth	-100<X<0	100
Seventh	-100<X<-20	120
Eighth	-100<X<-40	140
Ninth	-100<X<-60	160
Tenth	-100<X<-80	180

Table 1 – Item range coordinates relative to the Window.

As stated in the above table the regions are checked inside the timer method called timerPerformer placed in the MainAnimationApplet. On the other hand the Consumer equates the values 3 and 4 in the retrieve() method. The number 3 represents the item moving away from the buffer toward the consumer while the value 4 moves the remaining items in the buffer one slot forward since the item at the beginning was already removed. The range then adjusts accordingly, so when the producer inserts a new value it gets positioned in the correct location, the next available slot. The code that does this is show in the next page.

```
public void whichAnimationSequence(Graphics g)
{
        Graphics2D gg = (Graphics2D) g;

        if (theCorrectAnimationSequence==1)
        {
           theItemCoordinateY = ((Math.pow(theItemCoordinateX,3))*0.00009);
           theItemCoordinateX = (theItemCoordinateX+(theRequestedSpeed));
           theItemCoordinateY = (theItemCoordinateY * (-1));
        }

        if (theCorrectAnimationSequence==2)
        {
           theItemCoordinateX = theItemCoordinateX+(theRequestedSpeed);
        }


        if (theCorrectAnimationSequence==3)
        {
          theItemCoordinateW = ((Math.pow(theItemCoordinateQ,3))*0.00009);
          theItemCoordinateQ = (theItemCoordinateQ+(theRequestedSpeed));
          theItemCoordinateW = (theItemCoordinateW * (-1));
          theItemCoordinateX = theItemCoordinateQ+182;
          theItemCoordinateY = theItemCoordinateW;
        }

        if (theItemCoordinateX!=-100 && theItemCoordinateX!=-100)
        {
           buf2D.setColor(Color.red);
           buf2D.fill(new Rectangle2D.Double(theItemCoordinateX-
110,theItemCoordinateY,20,20));
           buf2D.setColor(Color.yellow);
           buf2D.draw(new Rectangle2D.Double(theItemCoordinateX-
110,theItemCoordinateY,20,20));
           buf2D.setFont(new Font("Arial", Font.PLAIN, 14));
           buf2D.drawString(""+theCorrectItemNo,(int)theItemCoordinateX-
103,(int)theItemCoordinateY+15);
        }


  }
```

Code Sample 4 – The Animation Sequence

Producer and Consumer Code was further amended to allow for termination when the STOP button is pressed. To accomplish this it was noted that a thread in our case the executing producer and consumer should arrange for their own death by having a run method that terminates naturally.

```
public void run()

{

    while(loopFOREVER)

    {
                if (loopFOREVER!=false)
                {
                    createRandomNumber();
                    theBoundedBuffer.enter(randomNumber);
                }
    }
}
```

Code Sample 5 – infinite thread loop

The above code runs indefinitely, although correct does not take into account the possibility of allowing the user to terminate. Therefore code must be added to take into account the ability for the user to stop the processes by activating the STOP button with the mouse.

"Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked ThreadDeath exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of stop should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the interrupt method should be used to interrupt the wait." [13]

```
public void run()
{

    while(loopFOREVER)
    {

        if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible)
        {
          loopFOREVER=false;
          theMainAnimationApplet.mainMenu.addBoundedBuffer.setEnabled(true);
          theMainAnimationApplet.mainMenu.changeSize.setEnabled(true);
        }


        if (loopFOREVER!=false)
        {
          createRandomNumber();
          theBoundedBuffer.enter(randomNumber);
        }


        if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible)
        {
          loopFOREVER=false;
          theMainAnimationApplet.mainMenu.addBoundedBuffer.setEnabled(true);
          theMainAnimationApplet.mainMenu.changeSize.setEnabled(true);
        }
    }

}
```

Code Sample 6 – The Thread Termination Technique

The amended code above, checks before any locks take place and after all the locks are released, therefore there is no possibility of inconsistencies in the values in the buffer. The variable theAnimationMustTerminateAsSoonAsPossible of type Boolean is defined in the class MainAnimationApplet and is adjusted to TRUE when the STOP button is pressed and back to FALSE when both threads terminate. It must be noted that when the STOP key is pressed any processes (either Producer or Consumer are in the CRITICAL REGION the termination would take place after this region is finished. All locks and unlocks remaining would take place naturally, but evidently the animation would be skipped since there would be conditional statements checking if the STOP button was activated.

The code in the Bounded Buffer was further extended in terms of code to deal with the REAL TIME issue that would allow the USER of the program to add/remove Producer/Consumer while the animation is in motion. An extra semaphore called deactivate of type binary was declared. In effect when the user clicked the area bounded by either the Producer or Consumer this would set a variable theProducerMustWAIT or theConsumerMustWait to TRUE if and only if the one clicked upon is not in its CRITICAL REGION in the case where variables producerHasPermissionToExit or consumerHasPermissionToExit are TRUE.

```
public void enter(int item)
{
        producerHasPermissionToExit=true;
        deActivate.pZ(0);
        // First Section of Thread
        mutex.P();
        if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
        {
                // Some code omiited, due to space limitations
        }
        mutex.V();
        empty.P();
        theMainAnimationApplet.whichMessageToDisplay=0;

        producerHasPermissionToExit=true;
        deActivate.pZ(1);
        // Second Section of Thread
         mutex.P();
         if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
         {
            theMainAnimationApplet.getTheThreadsCodeLocationAndDisplayIt(4,0);
         }
         mutex.V();

         producerHasPermissionToExit=true;
         deActivate.pZ(1);
         producerHasPermissionToExit=false;
         // Third Section of Thread: CRITICAL SECTION
         mutex.P();
         if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
         {
              // Some code omiited, due to space limitations
         }
         mutex.V();

         producerHasPermissionToExit=true;
         deActivate.pZ(0);
          // Fourth Section of Thread
         mutex.P();
         if (theMainAnimationApplet.theAnimationMustTerminateAsSoonAsPossible==false)
          {
              // Some code omiited, due to space limitations
          }
           mutex.V();
           full.V();

          producerHasPermissionToExit=true;
          deActivate.pZ(0);
    }
```

Code Sample 7 – The Thread Activation, De – Activation Technique

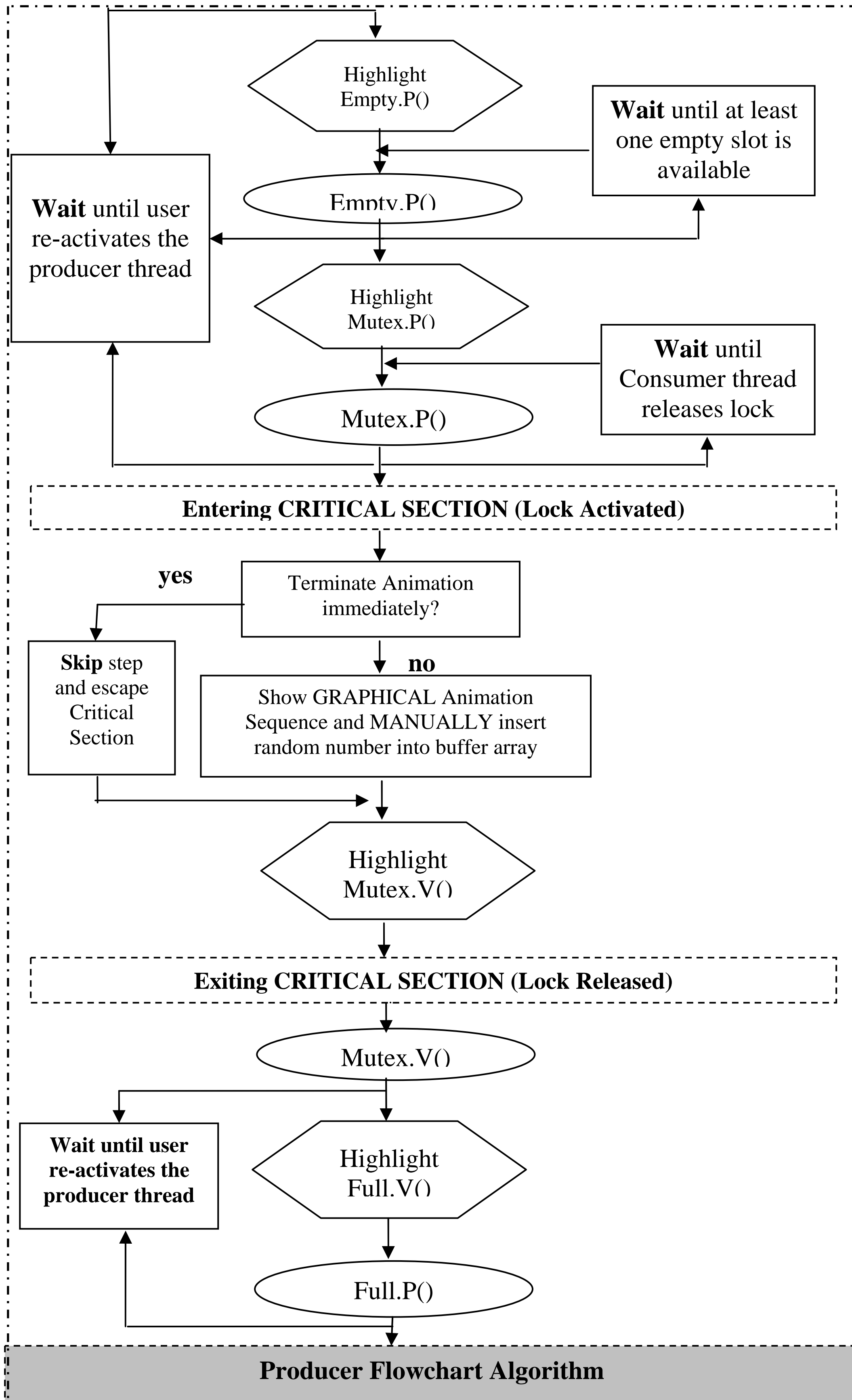


Figure 14 – Final Implemented Producer Algorithm

Summarizing, the main risk was that of time. Time was a major constraint and therefore needed to be contained. To do this the incremental approach was suitable in that it allowed the author to explore certain design criteria to examine the feasibility and suitability of each of these designs and to make the appropriate decisions whether to face the risk associated with each of them. In the case of the first design it seemed logical to move away from it, since it was forecasted that it could lead to more problems that of Applet restrictions in web browsers. There seemed to be an easier way to go about the project and it seemed that the second design would avoid obstacles such as those with the first one. 100% Java would allow for portability on any platform and the guaranteed ability to run under any browser. This turned out to be false, problems did arise but fortunately solutions were found considering that a lot of time had been lost at the beginning. The solutions involved a few designs in Java, those of single applet and multi applet. The final and successful one, was the variation of the single applet.

Chapter 7 - Testing

The purpose of this chapter is to describe the strategy used to produce and carry out test cases which would be used to see if the requirements outlined in Chapter 3 were fulfilled accurately.

Most of the testing was done during the implementation. This led to choosing the most suited design implementation for the project in terms of time constraints. Time constraint was the most important factor, because the project had begun at a very late stage.

Testing a software component is basically done to resolve the following issues [10].

- Check whether the component meets its specification and fulfils its functional requirements.
- Check whether the correct and complete structural and interaction requirements, specified before the development of the component, are reflected in the implemented software system.

Except for small programs, systems should not be tested as a single unit. Large systems are built out of sub-systems, which are built out of modules that are composed of procedures and functions. The testing process should therefore proceed in stages where testing is carried out incrementally in conjunction with system implementation. Black-box and white-box are test design methods.

- Black-box test design treats the system as a "black-box", so it doesn't explicitly use knowledge of the internal structure. Black-box test design is usually described as focusing on testing functional requirements.
- White-box test design allows one to peek inside the "box", and it focuses specifically on using internal knowledge of the software to guide the selection of test data.

7.1 Types of Testing Levels:

- **Unit testing:** Unit testing is Code oriented with Individual components tested to ensure that they operate correctly and therefore each component is tested independently, without other system components.

- **Module testing:** A module is a collection of dependent components such as an object class, an abstract data type or some looser collection of procedures and functions. A module encapsulates related components so it can be tested without other system modules.

- **Sub-system testing:** Also called Integration testing is more design oriented which involves testing collections of modules, which have been integrated into sub-systems. Sub-systems may be independently designed and implemented. The most common problems, which arise in large software systems, are sub-systems interface mismatches. The sub-system test process should therefore concentrate on the detection of interface errors by rigorously exercising these interfaces.

- **System testing:** The sub-systems are integrated to make up the entire system. The testing process is concerned with finding errors that result from unanticipated interactions between sub-systems and system components. It is also concerned with validating that the system meets its functional and non-functional requirements.

- **Acceptance testing:** This is the final stage in the testing process before the system is accepted for operational use. The system is tested with data supplied by the system client rather than simulated test data. Acceptance testing may reveal errors and omissions in the systems requirements definition because real data exercises the system in different ways from the test data. Acceptance testing may also reveal requirement problems where the system facilities do not really meet the user's needs (functional) or the system performance (non-functional) is unacceptable.

Acceptance testing is sometimes called alpha testing. Bespoke systems are developed for a single client. The alpha testing process continues until the system developer and the client agrees that the delivered system is an acceptable implementation of the system requirements. When a system is to be marketed as a software product, a testing process called beta testing is often used. Beta testing involves delivering a system to a number of potential customers who agree to use that system. They report problems to the system developers. This exposes the product to real use and detects errors that may not have been anticipated by the system builders. After this feedback, the system is modified and either released fur further beta testing.

7.2 Testing Techniques

There are many techniques that can be used to test software. Some are better than others, and some can be used in conjunction with others to get better test coverage. Some common testing techniques [10]:

- Manual testing - Tests are done by a human with test data that may be predetermined but may also be determined per test. In some cases, manual testing could be characterized as "banging away" at the software.
- Automated Testing - Tests can be run by a tool or an unattended process, such as a nightly build, and they can be re-run many times. Test data is predetermined or generated.
- Regression Testing - Tests, usually automated, are run to determine if modifications or enhancements have negatively affected functionality that passed previous testing.
- Stress Testing - Tests are run to determine if the software can withstand an unreasonable load with insufficient resources or extreme usage.
- Load Testing - Tests are run to determine if the software can handle a typical or maximum load or to determine the maximum sustainable load.
- Performance Testing - Tests are run to determine actual performance as compared to predicted performance.

The testing strategy taken mostly in this project was:

Testing began using the bottom-up technique which started with the fundamental components and worked upwards. At the modular level and worked outward towards the integration of the complete system in the form of a spiral. Unit testing started at the vortex of the spiral and concentrated on each unit of the software as implemented by the source code. In Integration testing the focus was on design and the production of the software architecture. Finally, in

System testing, the software as a whole and other system elements were tested together. Techniques used were manual regression and stress testing. No testing tools were used.

7.3 Program Test

7.3.1 Unit Testing

Some of the classes could not be unit tested, without certain alterations to the code, this being that the classes extended JApplet. This was the case in the design were three Applets were being produced and which included the Main Animation Class, the Main Menu class and the Main Control Panel class. A separate directory was created and these files were changed to extend JFrame, which would allow the use of the main() method which could be used to test the program at the command prompt.

Only certain classes could be unit tested.
- Bounded Buffer
- MainAnimationApplet
- MainMenu
- ControlPanel

Bounded Buffer did not need any alterations to the code, as it did not extend anything.
- Boundary conditions were tested.
- All error handling paths were tested.

The main method would initialize the buffer, and produce some values. This included more values than the buffer could hold to see if an error message was created. Then values were removed.

Error messages included:
- Indicating that the Buffer is full and that the Consumer should be added for the animation can continue.
- Indicating that the Buffer is empty and that the Producer should be added for the animation to continue.
- Indicating that no buffer exists.

MainAnimationApplet had to be changed from JApplet to JFrame. Testing required to, check if items graphically could simulate the movement of the item.
- Producer to Bounded Buffer
- Bounded Buffer to Consumer

This was the trickiest part of the testing. The class was tested under two microprocessors that of Pentium III and Pentium IV and the necessary adjustments were made in the timing method.

At first the timing was based on a variable being incremented inside the timerPerformer method. When the timer was between a certain range of number a certain animation sequence would take place. I.e the simulated movement of the item leaving the Producer and heading towards the bounded Buffer. When the timercount surpassed this value another sequence would take place. This worked fine on an identical CPU or with the graphics repainted

staying the same. On a different CPU the timings would not be synchronized with the sequence.

```
timerPerformer = new ActionListener()
{
     TimerCount++;

     if (theTimerCount<55)
    {
           if (theItemCoordinateY>0 || theItemCoordinateY==-100)
           {
                  theCorrectAnimationSequence=1;
                  repaint();
            }

 else
                  theTimerCount=55;
            }
     }
```

Code Sample 8 – The original timer method

The incremented value was then removed and in its place the x and y coordinates where used to check if the item was in the correct range and if so choose the appropriate sequence.

```
timerPerformer = new ActionListener()
 {
         public void actionPerformed(ActionEvent evt)
        {

              if (theCorrectAnimationSequence==1 || theCorrectAnimationSequence==2)
              {
                     if (theItemCoordinateY>0 || theItemCoordinateY==-100)
                     {

                       theCorrectAnimationSequence=1;
                       repaint();
                     }
              }
         }
}
```

Code Sample 9 – The amended Timer Method

The other classes:

- Producer
- Consumer
- Semaphore

These needed to be combined with one or more classes. Producer and Consumer needed to call bounded Buffer class which in turn needed to call Semaphore class.

7.3.2 *Integration (Sub-System) Testing*

First test:

Bounded Buffer Class
Semaphore Class

Second test:

Bounded Buffer Class
Producer Class
Consumer Class
Semaphore Class

Third test:

Bounded Buffer Class
Producer Class
Consumer Class
Semaphore Class
MainAnimationApplet Class

7.3.3 *System Testing*

1. Different processor speeds were tested. Pentium IV 1.6GHz and Pentium IIII 866MHz.
2. Varied levels of background activity were added .The processor was tied up with competing, irrelevant background tasks to check for effects on races and interrupt handling.
3. Error messages.

7.3.4 *Regression Testing*

Threading issues

On Windows NT constantly using the 'While' condition produced unpredicted results with the sequence of producer and consumer turns. This resulted in the producer and consumer activating one after the other. Therefore priorities settings, had no real effect as in Solaris 2. Researching about this problem lead to the replacement of the While condition with the use of Thread.sleep() which seemed to be cross platform solution.

Possible causes: Thread.yield(), 'While' statements leave the VM hogging as much of the CPU time as they can get. Most of these problems are related to bugs in the various VMs, whose intensive revision has produced evolving levels of functionality and stability.

Graphics Adjustments

Difference in user interface was experienced on the two testing platforms. The GUI did not come out looking as intended. When using a JTextField with a certain Font size on Windows, this did not produce the same results under Solaris. A number of tweaks needed to be done to resolve the problem. Increasing the size of the JTextField and Font used solved the issue.

Possible causes: Every VM on every platform has its own AWT bugs and display glitches, and these bugs change as the VM's are revised. Most of these need working around.

7.3.5 *Stress testing*

An interesting problem experienced and resolved, was the ability to allow the user to add/remove a Producer and/or Consumer in REAL TIME provided that depending which one was chosen to be removed was not in its CRITICAL REGION. The implementation at first instance worked fine until Stress Testing. More specifically when the USER continually clicked on either process constantly the program would get confused, leading the buffer to produce abnormal results as well as the animation to freeze.

The cause in this was [12]:

- The repaint() method is asynchronous. Every time the mouse was clicked the paint request would be send to the queue of the paint dispatching thread. If multiple calls to repaint() occur on a component before the initial repaint request is processed, the multiple requests may be collapsed into a single call to update(). The algorithm for determining when multiple requests should be collapsed is implementation-dependent.

The solution was to count the number of clicks. Using the getClickCount() this would check to see if the number of clicks did not exceed 1 click. If these exceeded 1 then they would be ignored. Therefore repaint would only be called once.

7.4 Test Cases

Test cases were created**,** executed. Below is a list of them with their corresponding results.

Animation Panel TEST				
Req. No			Pass/Fail	Comments
1	Can Producer Image be **added/removed** while animation is **NOT** in **operation**		PASS	In bottom left corner of the Animation Panel

2	Can Consumer Image be **added/removed** while animation is **NOT** in **operation**		PASS	In top right corner of the Animation Panel
3	Can Buffer Image be **added/removed** while animation is **NOT** in **operation**		PASS	Centre of the Animation Panel
4	Can Producer Image be **removed** while animation **IS** in operation and **NOT** in its **CRITICAL SECTION**		PASS	Producer can be added/removed in REAL TIME except in CRITICAL SECTION
5	Can Consumer Image be **removed** while animation **IS** in operation and **NOT** in its **CRITICAL SECTION**		PASS	Consumer can be added/removed in REAL TIME except in CRITICAL SECTION

Table 2 – Animation Panel Test Case

Menu System TESTING				
Req. No			Pass/Fail	Comments
	Producer Menu Item. Tick **activates /deactivates** when user chooses item. When animation IS/NOT in progress		PASS	Working Correctly
	Consumer Menu Item. Tick **activates /deactivates** when user chooses item. When animation IS/NOT in progress		PASS	Working Correctly
	Buffer Menu Item. Tick **activates /deactivates** when user chooses item. When animation is **NOT** in progress		PASS	Working Correctly
	USER cannot activate **BUFFER** menu item while animation **is in Progress**		PASS	Buffer menu item is not accessible while animation is in motion
	User can change the speed of the animation at all times		PASS	

	User can change the size of the buffer while animation is **NOT** progress		PASS	Working Correctly
	USER cannot change the size of the BUFFER while animation **is in progress**		PASS	Buffer SIZE menu item is not accessible while animation is in motion

Table 3 – Main Menu Test case

Button Panel TESTING				
Req. No			Pass/Fail	Comments
	START Button is **accessible** when animation is **NOT** in operation		PASS	Only if there is a Buffer on the screen.
	START Button is **NOT accessible** when animation is in operation		PASS	The button is not enabled.
	STOP Button is **NOT accessible** when animation is **NOT** in operation		PASS	The button is not accessible if the animation has not been started.
	STOP Button is **accessible** when animation is in operation		PASS	The button is accessible only when the animation is active
	RESET BUTTON		N/A	Was not implemented. The process of RESETTING was automated for the USERS convenience. SEE Design (Chapter 5)

Table 4 – Main Buttons Test Case

Control Panel TESTING				
Req. No			Pass/Fail	Comments
	Producer Priority		PASS	Works as predicted under Solaris 2 but not so in Windows
	Consumer			Works as

	Priority		PASS	predicted under Solaris 2 but not so in Windows. Its Operating System related and not software.
	Producers Code steps.		PASS	Work correctly
	Consumers Code steps		PASS	Work correctly

Table 5 – Control Panel Test Case

Chapter 8 – Critical Evaluation & Conclusions

The purpose of the chapter is to outline the positive and negative aspects of the project that occurred during the process of the development. An evaluation of the final product is done to see if the requirements were fulfilled as planned.

The original requirements as outlined in chapter 3 were not completely fulfilled. The main requirement which was not implemented was that of the UNIX POSIX standards which would require the use of C as the main language for accomplishing the task. The reason being as explained in Chapter 5 was due to interaction problems of the Java GUI with the C program. The assumption given to this problem was that it was thread related. Indeed the theory given, ended up being plausible during the development of the 100% Java design. During the design, implementation and testing stages the author realised that Java threading is not consistent on every platform as it claims to be. In this case SOLARIS 2 handles threads differently than Windows NT. That explained why the problem did not exist during testing of the Java & C interaction in the Windows operating system as outlined in Chapter 5. A plausible solution would have been to create the GUI on a separate thread and the C calls on another one.

The main reason for going on to the second design was that it was claimed that Java was platform independent and therefore would be ideal for using it to run on both operating systems on campus without any alterations in the code. This would then fulfil the second set of amended requirements. This turned out to be false. Problems arose during the development.

- Conditional "while" loops work differently on both operating systems in terms of threading.

The GUI had different outputs on the two operating systems. Fonts which displayed correctly in a JTextfield in Windows did not output the same on SOLARIS 2. Differences in platform user interface mean that some of the interface may not come out looking like as intended. Fortunately tweaking needs to be done on placements, sizes and other details in a platform neutral way, trying out the results on each of the target platforms is necessary as the project progresses. Layout managers should be used, not absolute locations

- Thread priorities which run correctly in SOLARIS 2 did not have the predicted results in Windows.
- JApplets working correctly in the applet viewer did not do so in the Web Browsers. Due to restrictions. Unfortunately the Java virtual machine often produces some pretty unhelpful error messages. For example, "class not found" could mean almost anything. It roughly translates as "something went wrong."

In the case of thread priorities, Java, in theory at least, provides ten levels. (If two or more threads are both waiting to run, the one with the highest priority level will execute.) In Solaris, which supports 2^{31} priority levels, this is not a problem (though Solaris priorities can be tricky to use). NT, on the other hand, has seven priority levels available, and these have to be mapped into Java's ten. This mapping is undefined therefore many possibilities present themselves. As an example, Java priority levels 1 and 2 might both map to NT priority level 1, and Java priority levels 8, 9, and 10 might all map to NT level 7.

In NT, priority levels are a problem if scheduling is required to be controlled. Even more complicated is the fact that priority levels aren't fixed. NT provides a mechanism called priority boosting [17], which can be turned off with a C system call, but not from Java. When priority boosting is enabled, NT boosts a thread's priority by an indeterminate amount of time when it executes certain I/O-related system calls. In practice, this means that a thread's priority level could be higher than originally speculated because that thread happened to perform an I/O operation at an inappropriate time.

The point of NT's priority boosting is to prevent threads that are doing background processing from impacting the apparent responsiveness of UI-heavy tasks. Other operating systems have more-sophisticated algorithms that typically lower the priority of background processes. The disadvantage of this mechanism, particularly when implemented on a per-thread rather than a per-process level, is that it's very difficult to use priority to determine when a particular thread will run.

In Solaris, as is the case in all UNIX variant systems, processes have priority as well as threads. The threads of high-priority processes can't be interrupted by the threads of low-priority processes. Moreover, the priority level of a given process can be limited by a system administrator so that a user process won't interrupt critical OS processes. NT does not support this feature. An NT process is just an address space. The system schedules threads; then, if a given thread is running under a process that isn't in memory, the process is swapped in. NT thread priorities fall into various "priority classes," that are distributed across a continuum of actual priorities. In essence a high-priority thread of an idle priority class process can pre-empt a low-priority thread of a normal priority class process, but only if that process is running in the background. NT provides no way to limit the priority class of a process. Any thread on any process on the machine can take over control of the box at any time by boosting its own priority class leading to no defence against this [16].

In practice, priority is virtually worthless under NT. Between NT's limited number of priority levels and it's uncontrollable priority boosting, there's no absolutely safe way for a Java program to use priority levels for scheduling. One workable compromise is to restrict to the option of using Thread.MAX_PRIORITY, Thread.MIN_PRIORITY, and Thread.NORM_PRIORITY when calling setPriority(). This restriction at least avoids the 10-levels-mapped-to-7-levels problem. Or another option is to use the os.name system property to detect NT, and then call a native method to turn off priority boosting, but that probably might not work on Explorer unless Sun's VM plug-in is used because Microsoft's VM uses a non standard native-method implementation. In any event, native methods can lead to more problems as discussed in the Java to C, JNI implementation mentioned in Chapter 4. To avoid the problem as much as possible, the best solution is to putt most threads at NORM_PRIORITY and using scheduling mechanisms other than priority.

This difference in underlying thread implementation means that assumptions about thread scheduling, priority or timing beyond what is specified in the Java API must be avoided. Threads of equal priority will get very different treatment depending on the VM.

Java is inconsistent.

- Different version of the VM exist - For that reason, it's important to keep in mind the possibility that some anomalous program behaviour that defies explanation may be a bug in a particular JVM. But not all Java platform dependence results from JVM implementation bugs. Significant platform dependence is introduced by the JVM specification itself. When the details of a JVM are left open at the specification level, it can produce vendor-dependent behaviour across JVMs

- JApplet running on an applet viewer does not necessarily mean that it will run on a web browser.

Although these problems arose during the implementation process, solutions were found by the author, some of which are described in Chapter 6, during the testing process. Therefore it can be concluded that the second set of requirements in Chapter 3 were successfully implemented. If more time was available the software would have been expanded to:

Producer/Consumer Problem:

- The ability for the user to add two or more:
  - Producers
  - Consumers

Philosophers Problem

- A visualization interface for the Philosopher software. (Due to time constraints this was not achieved). However, this could now be developed rapidly using the knowledge and technique gained through implementation of the Producer/Consumer software.

Also testing on more UNIX variants and an attempt to detect each operating system and execute a different GUI code to produce the same look and feel, without the need for tweaking using the code below:

```
String whichOS = System.getProperty("os.name");

if (whichOS.contains("UNIX"))
{
// do UNIX specific stuff here
}

if (whichOS.contains("Windows"))
{
// do Windows specific stuff here
}
```

Code Sample 10 – Determining the Operating System

The ability to use an external testing such as:

- JUnit.
- WINRUNNER for GUI testing.

Although the cost of writing cross-platform is much less with the Java language than in many other languages, it's not zero. The best advice is to run unit tests on as many platforms as possible, using as many JVM versions as possible. And, as always, avoid writing bug-prone code. Bug-prone code and platform dependence are a deadly combination.

Bibliography

[1] Andrew S. Tanenbaum, Albert S.WoodHull (1997) Operating Systems Design and Implementation, Second Edition, Prentice Hall.

[2] Avi Silberschatz, Peter Galvin, Greg Gagne (2000) Applied Operating System Concepts, First Edition, John Wiley & Sons, Inc.

[3] Fred Zlotnick (1991) The POSIX.1 Standard, A Programmer's Guide, The Benjamin/Cummings Publishing Company, Inc. California, USA

[4] Donald A. Lewine (1991) POSIX Programmer's Guide, O'Reilly & Associates, Inc, California, USA

[5] Milan Milenkovic (1992) Operating Systems Concepts & Design, McGraw-Hill International Editions, Second Edition, USA

[6] Ida M.Flynn/Ann Mclver McHoes (2001) Understanding Operating Systems, Bill Stenquist, Third Edition, USA

[7] *Biel School of Engineering,* An Incremental Software Development Process, http://www.hta-bi.bfh.ch/~due/se1/script/generated/sdp.fm.html

[8] Multithreading Models, Sun Microsystems Inc, http://docs.sun.com/db/doc/806-3461/6jck06gqk?a=view

[9] 1994-2004 Sun Microsystems, The Life Cycle of a Thread, http://java.sun.com/docs/books/tutorial/essential/threads/lifecycle.html

[10] Deb Stacey, Software Testing Techniques, University of Guelph, http://hebb.cis.uoguelph.ca/~dave/27320/testing/testing.html

[11] Dr. Eric Dubuis (1998-99), Software Engineering 1: A Software Design Process, http://www.hta-bi.bfh.ch/~due/99/c330/scripts/sdp/sdp_book.pdf

[12] Amy Fowler, Painting in AWT and Swing, Sun Microsystems Inc, http://java.sun.com/products/jfc/tsc/articles/painting/

[13] Threads Changes, Sun Microsystems Inc, http://java.sun.com/docs/books/tutorial/post1.0/preview/threads.html

[14] What Is a Thread? Sun Microsystems Inc, http://java.sun.com/docs/books/tutorial/essential/threads/definition.html

[15] Marcus Green, Java2 Certification Tutorial, Marcus Green Inc. 1999 http://www.jchq.net/tutorial/07_03Tut.htm

[16] Mark Russinovich, NT vs.UNIX: Is One Substantially Better, by Penton Media Inc http://www.winntmag.com/Articles/ArticleID/4500/pg/2/2.html

[17] Mark Russinovich, Inside the Windows NT Scheduler, by Penton Media Inc http://www.winnetmag.com/Articles/ArticleID/302/pg/2/2.html