GAGANDEEP SINGH

PRESENTS

The Dining Philosopher Problem

(A JAVA TECHNOLOGY PRODUCT)

Submitted in the partial fulfilment of the
Requirements for the award degree
Of
Bachelor Of Technology

Submitted By :

Gagandeep Singh
Enrollment No : 00413202709
CSE - 5th Semester

Department of Computer Science & Engineering
Guru Tegh Bahadur Institute of Technology

Guru Gobind Singh Indraprastha University
Kashmere Gate,New Delhi

Year 2011-2012

The Dining Philosopher Problem

Submitted in the partial fulfilment of the
Requirements for the award degree

Of

Bachelor Of Technology

Submitted By :

Gagandeep Singh
Enrollment No : 00413202709
CSE - 5th Semester

Department of Computer Science & Engineering
Guru Tegh Bahadur Institute of Technology

Guru Gobind Singh Indraprastha University
Kashmere Gate,New Delhi

Year 2011-2012

ACKNOWLEDGMENT

I express my deep sense of gratitude and obligation to 'CMC Limited (A Tata Enterprise)' and respected trainers for there valuable guidance, interest and constant encouragement given to me throughout Summer Training 2011, for the fulfillment of this project.

I am also grateful to my parents , my friends and my study institute Guru Tegh Bahadur Institute of Technology for providing me required material and helping in compiling the project.

August 2011 **Gagandeep Singh**

Preamble

The project titled ‘ The Dining Philosopher Problem ’ is a part of Summer Training in Java Programming Language 20 July - 8 August 2011 and has been submitted as a summer training project in Guru Tegh Bahadur Institute of Technology for partial fulfillment of the requirements for the award degree of Bachelor of Technology in CSE.
The source code has been programmed in Java and compiled in NetBeans IDE 6.9.1 with Java SE Development Kit 6.

The Project aims at developing a system of five processes competing for five resources in deadlock free environment. The project uses concept of process synchronization and java multithreading & swings as described in semester’s subjects.

Gagandeep Singh

CERTIFICATE

This is to certify that dissertation entitled

The Dining Philosopher Problem

, which is submitted by *Gagandeep Singh* in partial fulfillment of the requirement for the award of the degree of Bachelor of Technology in Computer Science & Engineering, Guru Tegh Bahadur Institute of Technology, New Delhi is an authentic record of the candidate's own work carried out during summer vacation 2011. The matter embodied in this thesis is original and has not been submitted for the award of any other degree.

Index

Introduction

The dining philosopher problem is considered a classic synchronization problem as it is an example of large class of concurrency-control problem. It is a simple representation of the need to allocate several resources among several processes in a deadlock-free and starvation-free manner. In 1965, Edsger Dijkstra set an examination question on a synchronization problem where five computers competed for access to five shared tape drive peripherals. Soon afterwards the problem was retold by Tony Hoare as the dining philosophers problem.

Aim : The aim of the project is to develop a system of five processes/threads (represented by five philosophers) that compete with each other for five resources (represented by chopsticks).

Objective : To demonstrates a deadlock situation and implement various technique to prevent it.

Problem Statement : Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five chopsticks. When a philosopher thinks, he does not interact with his colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him (the chopsticks that are between him and his left and right neighbors). A philosopher may pick up only one chopstick at a time. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he is finished eating, he puts down both of his chopsticks and start thinking again.

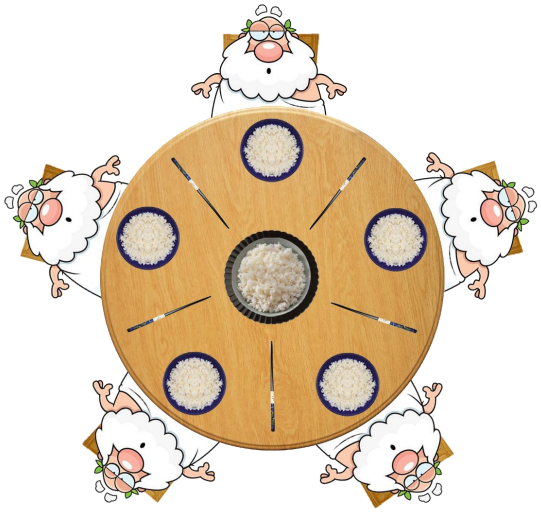


Fig 1 – Illustration of the dining philosophers problem

Issue : Suppose that all five philosophers become hungry simultaneously and grabs his left chopstick. When each philosopher tries to grab his right chopstick, he will be delayed forever.
Mutual exclusion is the core idea of the problem, and the dining philosophers create a generic and abstract scenario useful for explaining issues of this type. The failures these philosophers may experience are analogous to the difficulties that arise in real computer programming when multiple programs need exclusive access to shared resources.
The original problems of Dijkstra were related to external devices like tape drives. However, the difficulties studied in the Dining Philosophers problem arise far more often when multiple processes access sets of data that are being updated. Systems that must deal with a large number of parallel processes, such as operating system kernels, use thousands of locks and synchronizations that require strict adherence to methods and protocols if such problems as deadlock, starvation, or data corruption are to be avoided.

Solutions : The solutions to dining problem involves establishment of set of protocols that ensure a deadlock free environment. Some of the solutions demonstrated are :

- Allow at most four philosopher to be sitting simultaneously at the table.
- Allow a philosopher to pick up his chopsticks only if both chopsticks are available.
- Use an asymmetric solution i.e. an odd philosopher picks up his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick.

Theory

Process : A process is a program in execution i.e. a process is an active entity that includes the current activity, as represented by the value of the *program counter* and the *contents* of the processor's registers. A process may also include the *process stack*, containing temporary data (such as subroutine parameters, return addresses, and temporary variables), and a *data section* containing global variables.

Thread : A thread, sometimes called a *lightweight* process, is a basic unit of part of a process consisting of a program counter, a register set, and a stack space. A **thread of execution** is the smallest unit of processing that can be scheduled by an operating system. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variables reference at any given moment). To give an analogy, multiple threads in a process are like multiple cooks reading off the same cook book and following its instructions, not necessarily from the same page.

Threads differ from traditional processes in the fact that :

- Processes are typically independent, while threads exist as subsets of a process.
- Processes carry considerably more state information than threads, whereas multiple threads within a process share process state as well as memory and other resources.
- Processes have separate address spaces, whereas threads share their address space.
- Processes interact only through system-provided inter-process communication mechanisms
- Context switching between threads in the same process is typically faster than context switching between processes.

MultiThreading : Multithreading as a widespread programming and execution model allows multiple threads to exist within the context of a single process. These threads share the process' resources but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. However, perhaps the most interesting application of the technology is when it is applied to a *single* process to enable *parallel execution* on a *multiprocessor* system.

On a single processor, **multithreading** generally occurs by time-division multiplexing (as in multitasking): the processor switches between different threads. This context switching generally happens frequently enough that the user perceives the threads or tasks as running at the same time. On a multiprocessor or multi-core system, the threads or tasks will actually run at the same time, with each processor or core running a particular thread or task.

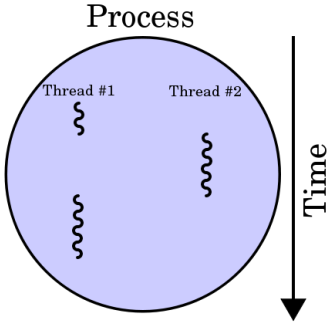


Fig 2- A process with two threads of execution on a single processor.

Operating systems schedule threads in one of two ways :

- *Preemptive multithreading* is generally considered the superior approach, as it allows the operating system to determine when a context switch should occur. The disadvantage to preemptive multithreading is that the system may make a context switch at an inappropriate time, causing lock convoy, priority inversion or other negative effects which may be avoided by cooperative multithreading.
- *Cooperative multithreading*, on the other hand, relies on the threads themselves to relinquish control once they are at a stopping point. This can create problems if a thread is waiting for a resource to become available.

Several States of a Thread :

- A thread can be *running*.
- It can be *ready* to run as soon as it gets CPU time.
- A running thread can be *suspended* , which temporarily suspends its activity.
- A suspended thread can then be *resumed,* allowing it to pick up where it left off.
- A thread can be *blocked* when waiting for a resource.
- At any time, a thread can be *terminated*, which halts its execution immediately. Once terminated, a thread cannot be resumed.

The parallel execution of threads in a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines — because the threads of the program naturally lend themselves to truly concurrent execution. In such a case, the programmer needs to be careful to avoid **race conditions**, and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to rendezvous in time in order to process the data in the correct order. Threads may also require **mutually-exclusive operations** (often implemented using semaphores) in order to prevent common data from being simultaneously modified, or read while in the process of being modified. Careless use of such primitives can lead to **deadlocks**.

Another use of multithreading, applicable even for single-CPU systems, is the ability for an application to remain responsive to input. In a single-threaded program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a *worker thread* that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep program responsive, and non-blocking I/O can be used to achieve the same result.

Process Synchronization : Process synchronization or serialization, strictly defined, is the application of particular mechanisms to ensure that two concurrently-executing threads or processes do not execute specific portions of a program at the same time. If one process has begun to execute a serialized portion of the program, any other process trying to execute this portion must wait until the first process finishes. Synchronization is used to control access to state both in small-scale multiprocessing systems -- in multithreaded and multiprocessor computers -- and in distributed computers consisting of thousands of units -- in banking and database systems, in web servers, and so on.

Race condition : A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place.

The Critical-Section Problem

Consider a system consisting of n processes {P_0 , P_1 , ……, P_{n-1}}. Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical section by the processes is *mutually exclusive* in time.
The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section.
The section of code implementing this request is the *entry section*. The critical section may be followed by an *exit section*. The remaining code is the *remainder*.

A solution to the critical-section problem must satisfy following three requirements :

- **Mutual Exclusion :** If the process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

- **Progress :** If no process is executing in its critical section and there exist some process that wish to enter their critical section, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this selection cannot be postponed indefinitely.

- **Bounded Waiting :** There exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

```
repeat

    entry section

        critical section

    exit section

        remainder section

until false
```

Fig 3 – A general structure of a typical process

Concept of Semaphores

The solutions to the critical-section problem are not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool, called semaphores.

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations – wait and signal. The classic definitions of wait and signal are :

```
wait(S)     :     while S ≤ 0
                     do no-op;
                  S := S - 1;

signal(S)   :     S := S + 1;
```

Modifications to the integer value of the semaphore in the `wait` and `signal` operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the `wait(S)`, the testing of the integer value S ($S \leq 0$), and its possible modification ($S := S - 1$), must also be executed without interruption.

Usage : we can use semaphores to deal with the n-process critical-section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1.
For example, consider two concurrently running processes: P_1 with a statement S_1 , and P_2 with a statement S_2 . Suppose that we require that S_2 be executed only after S_1 has completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore Synch, initialized to 0, and by inserting the statements

```
S1;
signal(synch);
```

in process P_1 and the statements

```
wait(synch);
S2;
```

in process P_2. Because synch is initialized to 0, P_2 will execute S_2 only after P_1 has invoked signal(synch), which is after S_1.

```
repeat

    wait(mutex);

        critical section

    signal(mutex);

        remainder section

until false
```

Fig 4 – Mutual-exclusion implementation with semaphores.

Concept of Monitors

Another high-level synchronization construct is the monitor type. A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consist of declarations of variables whose values define the sate of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type.
The syntax of monitor is :

```
type monitor-name = monitor
    variable declarations

    procedure entry P1(…);
       begin .... end;

    procedure entry P2(…);
       begin .... end;
            .
            .
            .
            .
    procedure entry Pn(…);
       begin .... end;

    begin
        initialization code
    end
```

The representation of a monitor type cannot be used directly by the various processes. Thus, a procedure defined a monitor can access only those variables declared locally within the monitor and formal parameters. Similarly, the local variables of a monitor can be accessed by only the local procedures.
The monitor construct ensure that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization constraint explicitly.

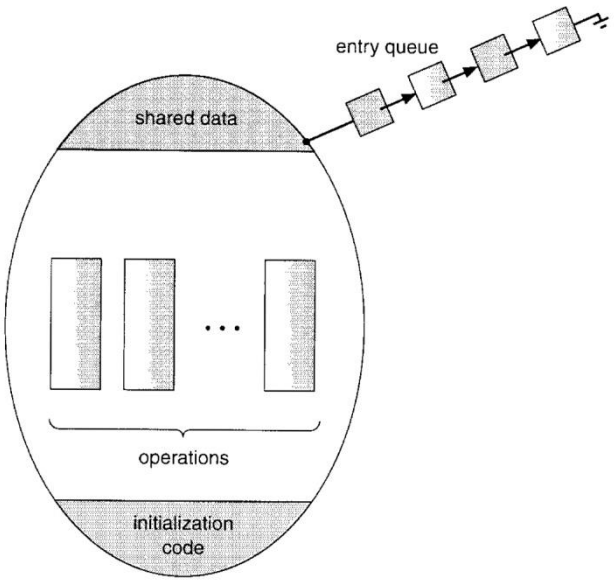


Fig 5 – Schematic view of a monitor

However, the monitor construct, as defined so far, is not sufficiently powerful for modeling synchronization mechanisms. These mechanisms are provided by the *condition* construct. A programmer who need to write her own tailor-made synchronization scheme can define one or more variables of type condition.

Deadlocks : Deadlock is a situation where two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting process. In computer science, **Coffman deadlock** refers to a specific condition when two or more processes are each waiting for the other to release a resource, or more than two processes are waiting for resources in a circular chain.
Deadlock is a common problem in multiprocessing where many processes share a specific type of mutually exclusive resource known as a *software lock* or *soft lock*. Computers intended for the *time-sharing* and/or *real-time* markets are often equipped with a *hardware lock* (or *hard lock*) which guarantees *exclusive access* to processes, forcing serialized access. Deadlocks are particularly troubling because there is no *general* solution to avoid (soft) deadlocks.

For example, if two people who are drawing diagrams, with only one pencil and one ruler between them. If one person takes the pencil and the other takes the ruler, a deadlock occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil to finish his work with the ruler. Neither request can be satisfied, so a deadlock occurs.

It is often seen in a paradox like the "chicken or the egg". Perhaps the best illustration of a deadlock can be drawn from a law passed by the Kansas Legislature, it said –

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Necessary Conditions for Deadlock

A deadlock situation can arise if the following four conditions hold simultaneously in a system :

- **Mutual Exclusion :** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource. The requesting process must be delayed until the resources has been released.

- **Hold and wait :** There must exist a process that is holding at least one resource and is waiting to acquire additional resource that are currently being held by other process.

- **No preemption** : Resources cannot be preempted i.e. a resource can be released only voluntarily by the process holding it, after that process has completed its task.

- **Circular wait :** There must exist a set $\{P_0 , P_1 , \ldots . P_n \}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1 , P_1 is waiting for a resource held by P2 …, P_n is waiting for a resource that is held by P_0 .

Methods for handling Deadlock

Principally, there are three different methods for dealing with the deadlock problem :

- We can use a protocol to ensure that the system will never enter a deadlock state.
- We can allow the system to enter a deadlock state and then recover.
- We can ignore the problem all together and pretend that deadlock never occur in the system.

(1) Deadlock Prevention :

The deadlock prevention is a set of methods for ensuring that at least one of the necessary conditions for deadlock cannot hold.

- **Mutual Exclusion :** The mutual-exclusion condition must hold for non-sharable resources. For example, a printer cannot be simultaneously shared by several processes. Sharable resources on other hand, do not require mutually exclusive access, and thus cannot be involved in a deadlock. For example, read-only files at the same time can be granted simultaneous access.

- **Hold and wait :** To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any resources.
  One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
  An alternative protocol allows a process to request resources only when the process has none.

- **No Preemption :** To ensure that this condition does not hold, we can us e the following protocol :
  If a process that is holding some resources request another resource that cannot be immediately allocated to it, the all resources currently being held are preempted and added to the list of resources for which the process is waiting. The process will restart only when it can regain its old resources, as well as the new ones.
  Alternatively, if a process request some resources, we first check whether they are available. If they are, we allocate them. If they are not available, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process. If the resources are either available or held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be preempted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting along with preempted resources.

- **Circular Wait :** The circular wait condition: Algorithms that avoid circular waits include "disable interrupts during critical sections", and "use a hierarchy to determine a partial ordering of resources" (where no obvious hierarchy exists, even the memory address of resources has been used to determine ordering) and Dijkstra's solution.

(2) Deadlock Avoidance:

The deadlock avoidance requires that the operating system be given in advance additional information concerning which resource a process will request and use during its lifetime.

For every resource request, the system sees if granting the request will mean that the system will enter an *unsafe* state, meaning a state that could result in deadlock. The system then only grants requests that will lead to *safe* states. In order for the system to be able to determine whether the next state will be safe or unsafe, it must know in advance at any time the number and type of all resources in existence, available, and requested. One known algorithm that is used for deadlock avoidance is the Banker's algorithm, which requires resource usage limit to be known in advance. However, for many systems it is impossible to know in advance what every process will request. This means that deadlock avoidance is often impossible.
Two other algorithms are Wait/Die and Wound/Wait, each of which uses a symmetry-breaking technique. In both these algorithms there exists an older process (O) and a younger process (Y). Process age can be determined by a timestamp at process creation time. Smaller time stamps are older processes, while larger timestamps represent younger processes.

	Wait/Die	**Wound/Wait**
O needs a resource held by Y	O waits	Y dies
Y needs a resource held by O	Y dies	Y waits

It is important to note that a process may be in an unsafe state but would not result in a deadlock. The notion of safe/unsafe states only refers to the ability of the system to enter a deadlock state or not. For example, if a process requests A which would result in an unsafe state, but releases B which would prevent circular wait, then the state is unsafe but the system is not in deadlock.

(3) Deadlock Detection :

Often, neither avoidance nor deadlock prevention may be used. Instead, deadlock detection and process restart are used by employing an algorithm that tracks resource allocation and process states, and rolls back and restarts one or more of the processes in order to remove the deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler or OS. Detecting the possibility of a deadlock *before* it occurs is much more difficult and is, in fact, *generally* undecidable, because the halting problem can be rephrased as a deadlock scenario. However, in *specific* environments, using *specific* means of locking resources, deadlock detection may be *decidable*. In the *general* case, it is not possible to distinguish between algorithms that are merely waiting for a very unlikely set of circumstances to occur and algorithms that will never finish because of deadlock.
Deadlock detection techniques include, but is not limited to *model checking*. This approach constructs a finite state-model on which it performs a progress analysis and finds all possible terminal sets in the model. These then each represent a deadlock.

LiveLock

A livelock is similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing. Livelock is a special case of resource starvation; the general definition only states that a specific process is not progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Livelock is a risk with some algorithms that detect and recover from deadlock. If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered. This can be avoided by ensuring that only one process (chosen randomly or by priority) takes action.

Platform or tools used

Programming Language : Java

Java is a programming language originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere." Java is currently one of the most popular programming languages in use, particularly for client-server web applications.

Features of Java

- **Simple :** Java was designed to be easy for the professional programmer to learn and use effectively. Assuming that one have some programming experience, he/she will not find Java hard to master. If you already understand the basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.

- **Secure :** Java achieved the protection from virus by confining an applet to the Java execution environment and not allowing it access to other parts of the computer. The ability to download applets with confidence that no harm will be done and that no security will be breached is considered by many to be the single most innovative aspect of Java.

- **Portable :** Portability is a major aspect of the Internet because there are many different types of computers and operating systems connected to it. If a Java program were to be run on virtually any computer connected to the Internet, there needed to be some way to enable that program to execute on different systems. For example, in the case of an applet, the same applet must be able to be downloaded and executed by the wide variety of CPUs, operating systems, and browsers connected to the Internet. It is not practical to have different versions of the applet for different computers. The *same* code must work on *all* computers. Therefore, some means of generating portable executable code was needed.

- **Object-Oriented :** Although influenced by its predecessors, Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects. Borrowing liberally from many seminal object-software environments of the last few decades, Java manages to strike a balance between the purist's "everything is an object" paradigm and the pragmatist's "stay out of my way" model. The object model in Java is simple and easy to extend, while primitive types, such as integers, are kept as high-performance nonobjects.

- **Robust :** The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees you from having to worry about many of the most common causes of programming errors. Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time.

- **Multithreaded :** Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

- **Architecture-Neutral :** A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was "write once; run anywhere, any time, forever." To a great extent, this goal was accomplished.

- **Interpreted and High Performance :** As described earlier, Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. The Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler.

- **Distributed :** Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports *Remote Method Invocation (RMI).* This feature enables a program to invoke methods across a network.

- **Dynamic :** Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java

environment, in which small fragments of bytecode may be dynamically updated on a running system.

Java as Object-Oriented Programming Language

All object-oriented programming languages provide mechanisms that help you implement the object-oriented model. They are encapsulation, inheritance, and polymorphism.

- **Encapsulation :** *Encapsulation* is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse. One way to think about encapsulation is as a protective wrapper that prevents the code and data from being arbitrarily accessed by other code defined outside the wrapper. Access to the code and data inside the wrapper is tightly controlled through a well-defined interface. In Java, the basis of encapsulation is the class. A *class* defines the structure and behavior (data and code) that will be shared by a set of objects. Each object of a given class contains the structure and behavior defined by the class, as if it were stamped out by a mold in the shape of the class. For this reason, objects are sometimes referred to as *instances of a class.* Thus, a class is a logical construct; an object has physical reality.

- **Inheritance :** *Inheritance* is the process by which one object acquires the properties of another object. This is important because it supports the concept of hierarchical classification. Most knowledge is made manageable by hierarchical (that is, top-down) classifications. For example, a Golden Retriever is part of the classification *dog,* which in turn is part of the *mammal* class, which is under the larger class *animal.* Without the use of hierarchies, each object would need to define all of its characteristics explicitly. However, by use of inheritance, an object need only define those qualities that make it unique within its class. It can inherit its general attributes from its parent. Thus, it is the inheritance mechanism that makes it possible for one object to be a specific instance of a more general case.

- **Polymorphism :** *Polymorphism* (from Greek, meaning “many forms”) is a feature that allows one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.
  More generally, the concept of polymorphism is often expressed by the phrase “one interface, multiple methods.” This means that it is possible to design a generic interface to a group of related activities. This helps reduce complexity by allowing the same interface to be used to specify a *general class of action.* It is the compiler’s job to select the *specific action* (that is, method) as it applies to each situation. You, the programmer, do not need to make this selection manually. You need only remember and utilize the general interface.

MultiThreaded Programming in Java

Unlike many other computer languages, Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.There are two distinct types of multitasking: processbased and thread-based. A process is, in essence,a program that is executing. Thus, process-based multitasking is the feature that allows your computer to run two or more programs concurrently. For example, process-

based multitasking enables you to run the Java compiler at the same time that you are using a text editor. In processbased multitasking, a program is the smallest unit of code that can be dispatched by the scheduler. In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

Multitasking threads require less overhead than multitasking processes. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly. Threads, on the other hand, are lightweight. They share the same address space and ooperatively share the same heavyweight process. Interthread communication is inexpensive, and context

switching from one thread to the next is low cost. While Java programs make use of processbased multitasking environments, process-based multitasking is not under the control of Java. However, multithreaded multitasking is.

The Java Thread Model

The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles. The value of a multithreaded environment is best understood in contrast to its counterpart.

Single-threaded systems use an approach called an *event loop* with *polling.* In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler. Until this event handler returns, nothing else can happen in the system. This wastes CPU time. It can also result in one part of a program dominating the system and preventing any other events from being processed. In general, in a singled-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire

program stops running.

The benefit of Java's multithreading is that the main loop/polling mechanism is eliminated. One thread can pause without stopping other parts of your program. For example, the idle time created when a thread reads data from a network or waits for user input can be utilized elsewhere. Multithreading allows animation loops to sleep for a second between each frame

without causing the whole system to pause. When a thread blocks in a Java program, only the single thread that is blocked pauses. All other threads continue to run.

Swings in Java

Swing did not exist in the early days of Java. Rather, it was a response to deficiencies present in Java's original GUI subsystem: the Abstract Window Toolkit. The AWT defines a basic set of controls, windows, and dialog boxes that support a usable, but limited graphical interface.

One reason for the limited nature of the AWT is that it translates its various visual components into their corresponding, platform-specific equivalents, or *peers*. This means that the look and feel of a component is defined by the platform, not by Java. Because the AWT components use native code resources, they are referred to as *heavyweight*. The use of native peers led to several problems. First, because of variations between operating systems, a component might look, or even act, differently on different platforms. This

potential variability threatened the overarching philosophy of Java: write once, run anywhere. Second, the look and feel of each component was fixed (because it is defined by the platform) and could not be (easily) changed. Third, the use of heavyweight components caused some frustrating restrictions. For example, a heavyweight component is always rectangular and opaque.

Swing Is Built on the AWT

Although Swing eliminates a number of the limitations inherent in the AWT, Swing *does not* replace it. Instead, Swing is built on the foundation of the AWT. This is why the AWT is still a crucial part of Java. Swing also uses the same event handling mechanism as the AWT. Therefore, a basic understanding of the AWT and of event handling is required to use Swing.

Two Key Swing Features

Swing was created to address the limitations present in the AWT. It does this through two key features: lightweight components and a pluggable look and feel. Together they provide an elegant, yet easy-to-use solution to the problems of the AWT. More than anything else, it is these two features that define the essence of Swing.

- **Swing Components Are Lightweight**
  With very few exceptions, Swing components are *lightweight*. This means that they are written entirely in Java and do not map directly to platform-specific peers. Because lightweight components are rendered using graphics primitives, they can be transparent, which enables nonrectangular shapes. Thus, lightweight components are more efficient and more flexible. Furthermore, because lightweight components do not translate into native peers, the look and feel of each component is determined by Swing, not by the underlying operating system. This means that each component will work in a consistent manner across all platforms.

- **Swing Supports a Pluggable Look and Feel**
  Swing supports a *pluggable look and feel* (PLAF). Because each Swing component is rendered by Java code rather than by native peers, the look and feel of a component is under the control of Swing. This fact means that it is possible to separate the look and feel of a component from the logic of the component, and this is what Swing does. Separating out the look and feel provides a significant advantage: it becomes possible to change the way that a component is rendered without affecting any of its other aspects. In other words, it is possible to "plug in" a new look and feel for any given component without creating any side effects in the code that uses that component. Moreover, it becomes possible to define entire sets of look-and-feels that represent different GUI styles. To use a specific style, its look and feel is simply "plugged in." Once this is done, all components are automatically rendered using that style.

The MVC Connection

In general, a visual component is a composite of three distinct aspects:

- The way that the component looks when rendered on the screen
- The way that the component reacts to the user
- The state information associated with the component

No matter what architecture is used to implement a component, it must implicitly contain these three parts. Over the years, one component architecture has proven itself to be exceptionally effective: *Model-View-Controller*, or MVC for short.

The MVC architecture is successful because each piece of the design corresponds to an aspect of a component. In MVC terminology, the *model* corresponds to the state information associated with the component. For example, in the case of a check box, the model contains a field that indicates if the box is checked or unchecked. The *view* determines how the component is displayed on the screen, including any aspects of the view that are affected by the current state of the model. The *controller* determines how the component reacts to the user. For example, when the user clicks a check box, the controller reacts by changing the model to reflect the user's choice (checked or unchecked). This then results in the view being updated. By separating a component into a model, a view, and a controller, the specific implementation of each can be changed without affecting the other two. For instance, different view implementations can render the same component in different ways without affecting the model or the controller.

Components and Containers

A Swing GUI consists of two key items: *components* and *containers*. However, this distinction is mostly conceptual because all containers are also components. The difference between the two is found in their intended purpose: As the term is commonly used, a *component* is an independent visual control, such as a push button or slider. A container holds a group of components. Thus, a container is a special type of component that is designed to hold other components. Furthermore, in order for a component to be displayed, it must be held within a container. Thus, all Swing GUIs will have at least one container. Because containers are components, a container can also hold other containers. This enables Swing to define what is called a *containment hierarchy*, at the top of which must be a *top-level container*.

NetBeans IDE

NetBeans refers to both a platform framework for Java desktop applications, and an integrated development environment (IDE) for developing with Java, JavaScript, PHP, Python, Groovy, C, C++, Scala, Clojure, and others.
The NetBeans IDE is written in Java and can run anywhere a compatable JVM is installed, including Windows, Mac OS, Linux, and Solaris. A JDK is required for Java development functionality, but is not required for development in other programming languages.
The NetBeans platform allows applications to be developed from a set of modular software components called *modules*. Applications based on the NetBeans platform (including the NetBeans IDE) can be extended by third party developers.

Integrated modules

These modules are part of the NetBeans IDE.

- **NetBeans Profiler**
  The NetBeans Profiler is a tool for the monitoring of Java applications: It helps developers find

memory leaks and optimize speed. Formerly downloaded separately, it is integrated into the core IDE since version 6.0.
The Profiler is based on a Sun Laboratories research project that was named JFluid. That research uncovered specific techniques that can be used to lower the overhead of profiling a Java application. One of those techniques is dynamic bytecode instrumentation, which is particularly useful for profiling large Java applications. Using dynamic bytecode instrumentation and additional algorithms, the NetBeans Profiler is able to obtain runtime information on applications that are too large or complex for other profilers. NetBeans also support Profiling Points that let you profile precise points of execution and measure execution time.

- **GUI design tool**
  Formerly known as *project Matisse*, the GUI design-tool enables developers to prototype and design Swing GUIs by dragging and positioning GUI components. The GUI builder also has built-in support for JSR 296 (Swing Application Framework), and JSR 295 (Beans Binding technology).

- **NetBeans JavaScript editor**
  The NetBeans JavaScript editor provides extended support for JavaScript, Ajax, and CSS. JavaScript editor features comprise syntax highlighting, refactoring, code completion for native objects and functions, generation of JavaScript class skeletons, generation of Ajax callbacks from a template; and automatic browser compatibility checks. CSS editor features comprise code completion for styles names, quick navigation through the navigator panel, displaying the CSS rule declaration in a List View and file structure in a Tree View, sorting the outline view by name, type or declaration order (List & Tree), creating rule declarations (Tree only), refactoring a part of a rule name (Tree only).

NetBeans IDE Download Bundles

Users can choose to download NetBeans IDE bundles tailored to specific development needs. Users can also download and install all other features at a later date directly through the NetBeans IDE. Some of these are :

- NetBeans IDE Bundle for Web and Java EE
- NetBeans IDE Bundle for Ruby
- NetBeans IDE Bundle for Java ME
- NetBeans IDE Bundle for PHP
- NetBeans IDE Bundle for JavaFX
- NetBeans IDE Starter Kit (DVD)

Adobe Photoshop

Adobe Photoshop is a graphics editing program developed and published by Adobe Systems Incorporated. Adobe Photoshop is released in two editions: **Adobe Photoshop**, and **Adobe Photoshop Extended**, with the Extended having extra 3D image creation, motion graphics editing, and advanced image analysis features. Adobe Photoshop Extended is included in all of Adobe's Creative Suite offerings except Design Standard, which includes the Adobe Photoshop edition.

Photoshop has ties with other Adobe software for media editing, animation, and authoring.
The **.PSD** (Photoshop Document), Photoshop's native format, stores an image with support for most imaging options available in Photoshop. These include layers with masks, color spaces, ICC profiles, transparency, text, alpha channels and spot colors, clipping paths, and duotone settings. This is in contrast to many other file formats (e.g. .EPS or .GIF) that restrict content to provide streamlined, predictable functionality. PSD format is limited to a maximum height and width of 30,000 pixels. **.PSB** (Photoshop Big) format, also known as "large document format" within Photoshop, is the extension of PSD format to images up to 300,000 pixels in width or height. That limit was apparently chosen somewhat arbitrarily by Adobe, not based on computer arithmetic constraints (it is not close to a significant power of two, as is 30,000) but for ease of software testing. PSD and PSB formats are documented.
Photoshop's primary strength is as a pixel-based image editor, unlike programs such as Macromedia FreeHand (now defunct), Adobe Illustrator, Inkscape or CorelDraw, which are vector-based image editors. However, Photoshop also enables the creation, incorporation, and manipulation of vector graphics through its Paths, Pen tools, Shape tools, Shape Layers, Type tools, Import command, and Smart object functions. Utilization of these tools and commands provides convenience when it is desirable to combine pixel-based and vector-based images in one Photoshop document because it can eliminate the necessity to visit more than one software program and transfer files between them.

CorelDraw

CorelDRAW is a vector graphics editor developed and marketed by Corel Corporation of Ottawa, Canada. It is also the name of Corel's Graphics Suite.
Several innovations to vector-based illustration originated with CorelDRAW: a node-edit tool that operates differently on different objects, fit text-to-path, stroke-before-fill, quick fill/stroke color selection palettes, perspective projections, mesh fills and complex gradient fills.
CorelDRAW differentiates itself from its competitors in a number of ways:
The first is its positioning as a graphics suite, rather than just a vector graphics program. A full range of editing tools allow the user to adjust contrast, color balance, change the format from RGB to CMYK, add special effects such as vignettes and special borders to bitmaps. Bitmaps can also be edited more extensively using Corel PhotoPaint, opening the bitmap directly from CorelDRAW and returning to the program after saving. It also allows a laser to cut out any drawings.
CorelDRAW is capable of handling multiple pages along with multiple master layers. Multipage documents are easy to create and edit and the Corel print engine allows for booklet and other imposition so even simple printers can be used for producing finished documents. One of the useful features for single and multi-page documents is the ability to create linked text boxes across documents that can be resized and moved while the text itself resets and flows through the boxes. Useful for creating and editing multi-article newsletters etc.
Smaller items, like business cards, invitations etc., can be designed to their final page size and imposed to the printer's sheet size for cost-effective printing. An additional print-merge feature (using a spreadsheet or text merge file) allows full personalization for many things like numbered raffle tickets, individual invitations, membership cards and more.

Project-System Design

The project is based on following rules and assumptions:

- The Dining Philosopher Problem describes a model of a system consisting of five threads represented by five philosophers, all running asynchronously.
- The Thread share a common data represented by five chopsticks. Each philosopher (Thread) competes to acquire two chopsticks (resources) closest to him i.e. chopsticks that are only between him and his left & right neighbors. The philosophers are not allowed to pick any other chopsticks than above ones.
- Table arrangement :

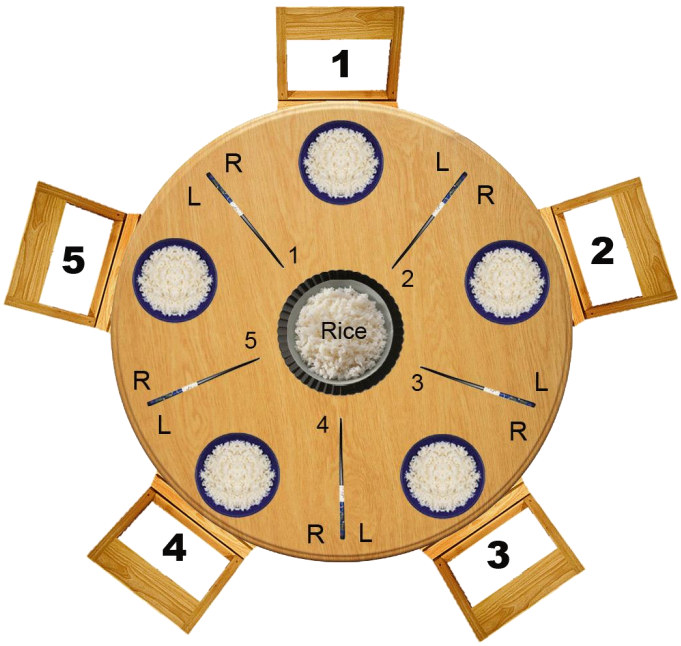


Fig 6 – Table arrangement

- All five philosophers sitting around a round table are uniquely labeled with an integer value ranging from 0 to 5 as shown above.
- Each chopstick lies between two philosophers and also has its own unique label number.
- Philosopher 1 has access to chopstick 1 and 2 only, Philosopher 2 has access to chopstick 2 and 3, Philosopher 3 to chopstick 3 and 4, Philosopher 4 to chopstick

4 and 5 and lastly Philosopher 5 has access to chopstick 5 and 1 only.

- States of a Philosopher :
    - 'Thinking' State : This is the initial state of a philosopher in which he simply thinks without any desire to eat or to perform any kind of action.
    - 'Waiting' State : This is the state where the philosopher waits to acquire chopsticks so that he can eat. A philosopher may be in waiting state if he wishes to eat and has either one or no chopsticks.
    - 'Eating' State : This state signifies that the philosopher has bath the chopsticks and is eating rice (executing).

- Rules for Philosophers :
    - All philosophers participating are initially at 'thinking' state where they think without desire to eat rice or own a chopstick.
    - Any philosopher who wishes to eat must first put into a 'waiting' state before he can perform any action like picking up chopsticks.
    - A philosopher may pick one chopstick at a time. (except in case of remedy 2 where he is allowed to pick both chopsticks)
    - A philosopher can change its state from waiting to eating (execute its body) if and only if he has two chopsticks at the same time. Else the philosopher will continue its state of 'waiting'.
    - Once in 'eating' state, a philosopher eats for 5 seconds and then releases both the chopsticks (release resources) so that they are made available for other philosophers.
    - Moreover, when a philosopher has completed eating once, he exits the dining hall which indicates that the thread has been removed from scheduling queue thereby freeing-up the memory. This also makes sure that the same philosopher does not get a chance to eat again and again thereby preventing **starvation.**

- **Critical Section** : 'Act of picking up a chopstick by philosopher is the critical section'. That is, the body of the code where each philosopher as a thread tries to pick up a chopstick is the critical section. This is because only one philosopher can pick a chopstick at a time. So, if two philosopher wishes to acquire a common chopstick then only that philosopher is awarded with the chopstick who has already entered his critical section.

- States of the system :
    - 'Safe' State : It is the desired state of the system that signifies that all philosophers competing with each other for chopsticks, respectfully get their chance to eat.
    - 'Deadlock' State : This state indicates an unsafe system, where the system goes into an endless loop since every philosopher with one chopstick each, is waiting for other one to release his chopstick which brings the system into a halt.

Rules for running the system :

- The system runs as an infinite loop until all philosophers participating get their chance to eat.
- Each loop performs three phases.
- At the start of the loop, all philosophers try to pick one chopstick available on the table.
- Next, all philosophers are again given a chance to pick a chopstick. Those who already have a chopstick seek for another one while those who are waiting empty handed seek for their first chopstick.
- Lastly, all those philosophers having two chopsticks eat for five seconds without releasing their chopsticks while others still remain in waiting state.
- The loop continues to perform one of the above three phase for respective philosophers till all of them leaves the dining hall.
- Once everybody has finished eating, the control is forcefully pulled out of the loop.

Event Rules :

- Case ‘deadlock’: All five philosophers are fired up randomly in an asynchronous manner such that each philosopher picks his left chopstick first and then goes for right chopstick in accordance to the iteration of the loop.
- Case ‘Remedy 1’ : In this solution for deadlock, only four philosophers are allowed to sit on the table instead of five. The system is then executed as per the above rule discussed.
- Case ‘Remedy 2’ : With all five philosophers, each one is allowed to pick his chopstick only if both are available. Even if one chopstick is available, it is not allocated to the demanding philosopher.
- Case ‘Remedy 3’ : In this case, a philosopher with odd label number pick his left chopstick first and then right chopstick whereas the philosopher with even label number picks his right chopstick first and then his left chopstick. The act of picking up chopstick is performed in accordance to the loop iteration.

Implementation and Coding

Chopstick.java

'Chopstick.java' file defines the real-world object 'chopstick'. The instance variable includes variables to specify chopstick label, its image and availability status. The class also includes certain 'set and get' methods for retrieving chopstick label & status and a method to reset the object chopstick to its initial state.

```
package diningphilosopher;

class Chopstick
{
    private int chpLabel;
    private boolean available;
    javax.swing.JLabel img;

    public Chopstick(int l)
    {
        chpLabel=l;
        available=true;
    }

    public void reset()
    {   img.setVisible(true);
        available=true;
    }

    public void setImage(javax.swing.JLabel i)
    { img=i; }

    public int getChopstickLabel()
    {  return chpLabel; }

    public boolean getChopstickStatus()
    {  return available; }

    public void setChopstickStatus(boolean a)
    {
       available=a;

       if(available==true)
       { img.setVisible(true);    }
       else
       { img.setVisible(false); }
```

```
    }

}
```

Philosopher.java

This file defines a class for the real-world entity 'philosopher' that includes instance variables for philosopher's label, status, his left & right chopstick, images for different state. The methods provides interface to access and modify various philosopher's state and his actions.

```
package diningphilosopher;

class Philosopher extends Thread
{
   private int phLabel;
   private int phStatus;
   private Chopstick chpLeft;
   private Chopstick chpRight;
   public boolean victim;

   javax.swing.JLabel img_think;
   javax.swing.JLabel img_wt_0;
   javax.swing.JLabel img_wt_L;
   javax.swing.JLabel img_wt_R;
   javax.swing.JLabel img_eat;
   javax.swing.JLabel img_tim;

   public Philosopher(int l,int s)
   {
       phLabel=l;
       phStatus=s;
       victim=false;
       chpLeft=null;
       chpRight=null;
   }

   public void reset()
   {
       phStatus=0;
       victim=false;
       chpLeft=null;
       chpRight=null;

       resetImage();
       img_think.setVisible(true);
   }
```

```
   private void resetImage()
   {
       img_think.setVisible(false);
       img_wt_0.setVisible(false);
       img_wt_L.setVisible(false);
       img_wt_R.setVisible(false);
       img_eat.setVisible(false);
       img_tim.setVisible(false);
   }

   public void setImgae(javax.swing.JLabel it, javax.swing.JLabel iw0 ,
javax.swing.JLabel iwl, javax.swing.JLabel iwr, javax.swing.JLabel ie,
javax.swing.JLabel itm)
   {
       img_think=it;
       img_wt_0=iw0;
       img_wt_L=iwl;
       img_wt_R=iwr;
       img_eat=ie;
       img_tim=itm;
   }


   public void allocatePhChopsticks(Chopstick cL,Chopstick cR,int rule)
   {

       switch(rule)
       {
           case 0: {
                       if( (chpLeft==null && chpRight==null) || (chpLeft==null
&& chpRight!=null) )
                       {  SelectSlot('L', cL);      }
                       else if(chpLeft!=null && chpRight==null)
                       {  SelectSlot('R', cR);      }
                   }break;

           case 1: {
                       if( (chpLeft==null && chpRight==null) || (chpLeft==null
&& chpRight!=null) )
                       {  SelectSlot('L', cL);      }
                       else if(chpLeft!=null && chpRight==null)
                       {  SelectSlot('R', cR);      }
                   }break;

           case 2: {
                       if( cL.getChopstickStatus()==true &&
cR.getChopstickStatus()==true )
                       {
                           SelectSlot('L', cL);
                           SelectSlot('R', cR);
                       }
                   }break;

           case 3: {
```

```
                        if(phLabel%2==0)       //even
                        {
                            if(chpLeft==null && chpRight==null)
                            {  SelectSlot('R', cR);      }
                            else if(chpLeft==null && chpRight!=null)
                            {  SelectSlot('L', cL);      }

                        }
                        else        //odd
                        {
                            if(chpLeft==null && chpRight==null)
                            {  SelectSlot('L', cL);      }
                            else if(chpLeft!=null && chpRight==null)
                            {  SelectSlot('R', cR);      }
                        }
                    }break;

        }
    }

    private void SelectSlot(char chNo,Chopstick c)
    {
        switch(chNo)
        {
            case 'L': {   if(c.getChopstickStatus()==true)
                        {  chpLeft = c;                     //allocate
                           c.setChopstickStatus(false);     // change chopstick
status
                           upgradePhStatus();
                           return;
                        }
                        else
                        { return;  }                // cannot allocate

                    }

            case 'R': {   if(c.getChopstickStatus()==true)
                        {  chpRight = c;
                           c.setChopstickStatus(false);
                           upgradePhStatus();
                           return;
                        }
                        else
                        { return;}
                    }
        }
     }

    private void releasePhChopsticks()
    {

        chpLeft.setChopstickStatus(true);
        chpLeft=null;
        chpRight.setChopstickStatus(true);
```

```
        chpRight=null;

        phStatus=0;    //THINKING
    }


    public void upgradePhStatus()
    {
       if(chpLeft==null && chpRight==null)
       { phStatus=1;
         try{  sleep(400); }
         catch(InterruptedException e)
         { System.out.println("Error in Sleep()"); }

         resetImage();
         img_wt_0.setVisible(true);
       }
       else if(chpLeft != null && chpRight == null)
       { phStatus=1;
         resetImage();
         img_wt_L.setVisible(true);
       }
       else if(chpLeft==null && chpRight!=null)
       { phStatus=1;
         resetImage();
         img_wt_R.setVisible(true);
       }
       else
       { phStatus=2;
         resetImage();
         img_wt_L.setVisible(true);
         img_wt_R.setVisible(true);
       }

    }

    public void performAction()
    {
       switch(phStatus)
       {
           case 0: {System.out.println("Philosopher "+phLabel+" is
'THINKING'");
                   }break;

           case 1: {System.out.println("Philosopher "+phLabel+" is 'WAITING'");
                   }break;

           case 2: {System.out.println("Philosopher "+phLabel+" is 'EATING'");


                           resetImage();
                           img_eat.setVisible(true);
                           img_tim.setVisible(true);
                           try
```

```
                            { sleep(5000); }
                            catch(InterruptedException e)
                            { System.out.println("Error in Sleep()"); }

                            victim=true;
                            releasePhChopsticks();
                            img_eat.setVisible(false);
                            img_tim.setVisible(false);

                    }break;
       }

    }

    public int getPhLabel()
    { return phLabel;    }

    public int getPhStatus()
    { return phStatus;    }

    public Chopstick getPhChp(char cno)
    {
        if(cno=='L')
        {  return chpLeft;   }
        else
        {  return chpRight;  }
    }

    public void deactivatePH()
    {  phStatus=-1;
       victim=true;
    }

}
```

Threads.java

The ‘Thread.java’ file defines a class named ‘Threads’ that inherits/extends pre-defined class ‘Thread’ using which threads for each philosopher are instantiated. Thus, every philosopher is a thread himself. The class defines instance variables for a specific philosopher and all its resources that he needs.

```
package diningphilosopher;

class Threads extends Thread
{
    private Philosopher thPh;
    private Chopstick thCL,thCR;
    private char ThFunc;
```

```
    private int rl;
    javax.swing.JLabel chpImg;

    public Threads(Philosopher p,Chopstick cL,Chopstick cR,char f,int r)
    {
        this.thPh=p;
        this.thCL=cL;
        this.thCR=cR;
        this.ThFunc=f;
        rl=r;
        start();
    }


    public void AllocateMethod()
    {
           thPh.allocatePhChopsticks(thCL, thCR,rl);
    }

    public void ExecuteMethod()
    {
           thPh.performAction();
    }

    public void run()
    {
        if(ThFunc=='A')
        { AllocateMethod();      }
        else if(ThFunc=='E')
        { ExecuteMethod();    }

    }
}
```

The file includes a class for creating a JFrame that displays the problem statement of the project in a separate window. The frame also includes a button for closing the JFrame.

PrbStScreen.java

```
package diningphilosopher;

public class PrbStScreen extends javax.swing.JFrame {

    public PrbStScreen() {
        initComponents();
    }

    @SuppressWarnings("unchecked")
```

```
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        jButton1 = new javax.swing.JButton();
        jLabel1 = new javax.swing.JLabel();


setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("Problem Statement");
        setAlwaysOnTop(true);
        setBounds(new java.awt.Rectangle(70, 70, 700, 620));
        setMinimumSize(new java.awt.Dimension(700, 620));
        setResizable(false);
        getContentPane().setLayout(null);

        jButton1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/CloseWin_butt
on_1.gif"))); // NOI18N
        jButton1.setBorder(null);
        jButton1.setBorderPainted(false);
        jButton1.setContentAreaFilled(false);
        jButton1.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        jButton1.setPressedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/CloseWin_butt
on_3.gif"))); // NOI18N
        jButton1.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/CloseWin_butt
on_2.gif"))); // NOI18N
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });
        getContentPane().add(jButton1);
        jButton1.setBounds(280, 550, 130, 40);

        jLabel1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/prb_stmt_bg.j
pg"))); // NOI18N
        jLabel1.setText("jLabel1");
        getContentPane().add(jLabel1);
        jLabel1.setBounds(0, 0, 700, 600);

        pack();
    }// </editor-fold>//GEN-END:initComponents

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton1ActionPerformed
        dispose();
    }//GEN-LAST:event_jButton1ActionPerformed

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
```

```
            public void run() {
                new PrbStScreen().setVisible(true);
            }
        });
    }

    // Variables declaration
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel1;
    // End of variables declaration
}
```

The file includes a class for creating another JFrame that displays the description about the project. The frame also includes a button for closing the JFrame.

AboutScreen.java

```
package diningphilosopher;

public class AboutScreen extends javax.swing.JFrame {

    public AboutScreen() {
        initComponents();
    }

    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        jButton1 = new javax.swing.JButton();
        jLabel1 = new javax.swing.JLabel();


setDefaultCloseOperation(javax.swing.WindowConstants.DISPOSE_ON_CLOSE);
        setTitle("About");
        setAlwaysOnTop(true);
        setBounds(new java.awt.Rectangle(50, 50, 400, 325));
        setMinimumSize(new java.awt.Dimension(400, 325));
        setResizable(false);
        getContentPane().setLayout(null);

        jButton1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/close_button_
1.gif"))); // NOI18N
        jButton1.setBorder(null);
        jButton1.setBorderPainted(false);
        jButton1.setContentAreaFilled(false);
```

```
        jButton1.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        jButton1.setFocusPainted(false);
        jButton1.setPressedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/close_button_
3.gif"))); // NOI18N
        jButton1.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/close_button_
2.gif"))); // NOI18N
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });
        getContentPane().add(jButton1);
        jButton1.setBounds(40, 250, 80, 30);

        jLabel1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/About_screen.
gif"))); // NOI18N
        getContentPane().add(jLabel1);
        jLabel1.setBounds(0, 0, 400, 300);

        pack();
    }// </editor-fold>//GEN-END:initComponents

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton1ActionPerformed
        // TODO add your handling code here:
        dispose();
    }//GEN-LAST:event_jButton1ActionPerformed

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new AboutScreen().setVisible(true);
            }
        });
    }

    // Variables declaration
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel1;
    // End of variables declaration
}
```

Screen1.java

The file defines a JFrame named 'Screen1' that displays introduction window of the project. The frame includes a button that closes this frame and initiates another JFrame that describes the system environment.

```
package diningphilosopher;

public class Screen1 extends javax.swing.JFrame {

    public Screen1() {
        initComponents();
    }

    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        jButton1 = new javax.swing.JButton();
        jLabel1 = new javax.swing.JLabel();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setTitle("The Dining Philosopher Problem | (c) 2011Gagandeep Singh");
        setAlwaysOnTop(true);
        setBounds(new java.awt.Rectangle(0, 0, 900, 700));
        setMinimumSize(new java.awt.Dimension(900, 700));
        setResizable(false);
        getContentPane().setLayout(null);

        jButton1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Click_button_
1.gif"))); // NOI18N
        jButton1.setBorder(null);
        jButton1.setBorderPainted(false);
        jButton1.setContentAreaFilled(false);
        jButton1.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        jButton1.setPressedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Click_button_
3.gif"))); // NOI18N
        jButton1.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Click_button_
2.gif"))); // NOI18N
        jButton1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                jButton1ActionPerformed(evt);
            }
        });
        getContentPane().add(jButton1);
        jButton1.setBounds(300, 500, 320, 60);
```

```
        jLabel1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/screen1_bg.jp
g"))); // NOI18N
        getContentPane().add(jLabel1);
        jLabel1.setBounds(0, 0, 900, 675);

        pack();
    }// </editor-fold>//GEN-END:initComponents

    private void jButton1ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_jButton1ActionPerformed
        dispose();
        new Screen2().setVisible(true);
    }//GEN-LAST:event_jButton1ActionPerformed

    public static void main(String args[]) {
        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                new Screen1().setVisible(true);
            }
        });
    }

    // Variables declaration
    private javax.swing.JButton jButton1;
    private javax.swing.JLabel jLabel1;
    // End of variables declaration

}
```

Screen2.java

The Main Class : This file defines the main class 'Screen2' of the project which describes the complete running system. The class extends a JFrame and provides a window environment for running the system using all previously defined classes.

```
package diningphilosopher;

public class Screen2 extends javax.swing.JFrame {

    //----------------DECLARATIONS-------------------------
        //Chopsticks
        static Chopstick c1=new Chopstick(1);
        static Chopstick c2=new Chopstick(2);
        static Chopstick c3=new Chopstick(3);
        static Chopstick c4=new Chopstick(4);
        static Chopstick c5=new Chopstick(5);

        //array for philosophers
        static Philosopher p[];
```

```
        //Thread declarations
        static Thread thAllChp[];
        static Thread thExPh[];

        //Variables
        static int ruleChoice=-1;              // For rule Choice
        static int noOfPh;                    // For total number of
philosophers
        static boolean DEADLOCK=true;          // Deadlock chk
        static boolean loop=true;
        static boolean exe=false;

    public Screen2()    //Creates new form Screen2
    {
        initComponents();

        //reset interface images
        deadlock_image.setVisible(false);
        stable_image.setVisible(false);
        start_deactive.setVisible(false);
        reset_button.setVisible(false);

        //reset philosophers images
        ph1_wt_0.setVisible(false);
        ph1_wt_L.setVisible(false);
        ph1_wt_R.setVisible(false);
        ph1_eat.setVisible(false);
        ph1_timmer.setVisible(false);

        ph2_wt_0.setVisible(false);
        ph2_wt_L.setVisible(false);
        ph2_wt_R.setVisible(false);
        ph2_eat.setVisible(false);
        ph2_timmer.setVisible(false);

        ph3_wt_0.setVisible(false);
        ph3_wt_L.setVisible(false);
        ph3_wt_R.setVisible(false);
        ph3_eat.setVisible(false);
        ph3_timmer.setVisible(false);

        ph4_wt_0.setVisible(false);
        ph4_wt_L.setVisible(false);
        ph4_wt_R.setVisible(false);
        ph4_eat.setVisible(false);
        ph4_timmer.setVisible(false);

        ph5_wt_0.setVisible(false);
        ph5_wt_L.setVisible(false);
        ph5_wt_R.setVisible(false);
        ph5_eat.setVisible(false);
        ph5_timmer.setVisible(false);
```

```
        sad_ph5.setVisible(false);


        //allocate image to chopstick objects
        c1.setImage(chp1_image);
        c2.setImage(chp2_image);
        c3.setImage(chp3_image);
        c4.setImage(chp4_image);
        c5.setImage(chp5_image);

        p=new Philosopher[5];     // Allocating Philosopher array
        p[0]=new Philosopher(1,0);  p[0].setImgae(ph1_thinking, ph1_wt_0,
ph1_wt_L, ph1_wt_R, ph1_eat, ph1_timmer);
        p[1]=new Philosopher(2,0);  p[1].setImgae(ph2_thinking, ph2_wt_0,
ph2_wt_L, ph2_wt_R, ph2_eat, ph2_timmer);
        p[2]=new Philosopher(3,0);  p[2].setImgae(ph3_thinking, ph3_wt_0,
ph3_wt_L, ph3_wt_R, ph3_eat, ph3_timmer);
        p[3]=new Philosopher(4,0);  p[3].setImgae(ph4_thinking, ph4_wt_0,
ph4_wt_L, ph4_wt_R, ph4_eat, ph4_timmer);
        p[4]=new Philosopher(5,0);  p[4].setImgae(ph5_thinking, ph5_wt_0,
ph5_wt_L, ph5_wt_R, ph5_eat, ph5_timmer);

        //comments
        hideAllComments();
        cmt_welcom.setVisible(true);
        cmt_select.setVisible(true);

    }

    public static void EXECUTEALGO()
    {

        DEADLOCK=true;

        switch(ruleChoice)
        {
            case 0: {  comment(6);

                       noOfPh=5;                        // Total Philosophers

                       thAllChp=new Thread[noOfPh];
                       thExPh=new Thread[noOfPh];


                       //set status to waiting
                       for(int i=0;i<noOfPh;i++)
                       {  p[i].upgradePhStatus();        }


                       do
                       {

                      //Threads for allocating chopsticks
```

```
                        //Allocate first chopstick
                        for(int j=0;j<noOfPh;j++)
                        {
                           if(p[j].victim!=true)     // check for starvation
                           {
                              switch(p[j].getPhLabel())
                              {
                                  case 1: thAllChp[j]=new
Threads(p[j],c2,c1,'A',ruleChoice); break;
                                  case 2: thAllChp[j]=new
Threads(p[j],c3,c2,'A',ruleChoice); break;
                                  case 3: thAllChp[j]=new
Threads(p[j],c4,c3,'A',ruleChoice); break;
                                  case 4: thAllChp[j]=new
Threads(p[j],c5,c4,'A',ruleChoice); break;
                                  case 5: thAllChp[j]=new
Threads(p[j],c1,c5,'A',ruleChoice); break;
                              }

                              try{
                                    thAllChp[j].sleep(1000);
                                    thAllChp[j].join();
                              }catch(InterruptedException e)
                              { System.out.println("Error in Sleep()"); }
                           }

                        }

                     //Wait for threads to finish
                     try
                     {
                          for(int j=0;j<noOfPh;j++)
                          {  thAllChp[j].join();    }
                     }catch(InterruptedException e)
                     { System.out.println("Main Interrupted!");  }


                     //Allocate Second chopstick
                        for(int j=0;j<noOfPh;j++)
                        {
                           if(p[j].victim!=true)     // check for
startvation
                           {
                              switch(p[j].getPhLabel())
                              {
                                  case 1: thAllChp[j]=new
Threads(p[j],c2,c1,'A',ruleChoice); break;
                                  case 2: thAllChp[j]=new
Threads(p[j],c3,c2,'A',ruleChoice); break;
                                  case 3: thAllChp[j]=new
Threads(p[j],c4,c3,'A',ruleChoice); break;
                                  case 4: thAllChp[j]=new
Threads(p[j],c5,c4,'A',ruleChoice); break;
```

```
                                    case 5: thAllChp[j]=new
Threads(p[j],c1,c5,'A',ruleChoice); break;

                                }
                                try{
                                      thAllChp[j].sleep(1000);
                                      thAllChp[j].join();
                                }catch(InterruptedException e)
                                { System.out.println("Error in Sleep()"); }
                               }


                           }

                        //Wait for threads to finish
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thAllChp[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }




                        //Threads for executing philosophers
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim!=true)         // check for
startvation
                            {  thExPh[j]=new Threads(p[j], null, null,
'E',ruleChoice);
                            }
                        }

                        //Wait for thread to finish
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thExPh[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }



                        //Check for deadlock
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim==true)
                            {  DEADLOCK=false; break; }
                            else
                            {  DEADLOCK=true;  }
                        }
                        if(DEADLOCK==true)
```

```
                        {  deadlock_image.setVisible(true); comment(3); }


                        //Check for loop
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim==false)
                            {  loop=true; break; }
                            else
                            {  loop=false;  }
                        }


                        }while(loop==true && DEADLOCK==false);

                        if(DEADLOCK==false)
                        {  stable_image.setVisible(true); comment(4); }


                    }break;

              case 1: {  comment(6);//4 philosopher sitting at a time

                        noOfPh=4;

                        //hide philosopher5
                        ph5_thinking.setVisible(false);
                        sad_ph5.setVisible(true);

                        thAllChp=new Thread[noOfPh];
                        thExPh=new Thread[noOfPh];


                        //set status to waiting
                        for(int i=0;i<noOfPh;i++)
                        {  p[i].upgradePhStatus();        }

                        do
                        {

                       //Threads for allocating chopsticks
                           //Allocate first chopstick
                           for(int j=0;j<noOfPh;j++)
                           {
                              if(p[j].victim!=true )     // check for
startvation
                              {
                                 switch(p[j].getPhLabel())
                                 {
                                     case 1: thAllChp[j]=new
Threads(p[j],c2,c1,'A',ruleChoice); break;
                                     case 2: thAllChp[j]=new
Threads(p[j],c3,c2,'A',ruleChoice); break;
```

```
                                    case 3: thAllChp[j]=new
Threads(p[j],c4,c3,'A',ruleChoice); break;
                                    case 4: thAllChp[j]=new
Threads(p[j],c5,c4,'A',ruleChoice); break;
                                    //case 5: thAllChp[j]=new
Threads(p[j],c1,c5,'A',ruleChoice); break;
                                }
                                try{
                                      thAllChp[j].sleep(1000);
                                      thAllChp[j].join();
                                }catch(InterruptedException e)
                                { System.out.println("Error in Sleep()"); }
                               }

                           }

                        //Wait for threads to finish
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thAllChp[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }


                        //Allocate Second chopstick
                           for(int j=0;j<noOfPh;j++)
                           {
                              if(p[j].victim!=true)     // check for
startvation
                              {
                                 switch(p[j].getPhLabel())
                                 {
                                     case 1: thAllChp[j]=new
Threads(p[j],c2,c1,'A',ruleChoice); break;
                                     case 2: thAllChp[j]=new
Threads(p[j],c3,c2,'A',ruleChoice); break;
                                     case 3: thAllChp[j]=new
Threads(p[j],c4,c3,'A',ruleChoice); break;
                                     case 4: thAllChp[j]=new
Threads(p[j],c5,c4,'A',ruleChoice); break;
                                     //case 5: thAllChp[j]=new
Threads(p[j],c1,c5,'A',ruleChoice); break;
                                 }
                                 try{
                                       thAllChp[j].sleep(1000);
                                       thAllChp[j].join();
                                 }catch(InterruptedException e)
                                 { System.out.println("Error in Sleep()"); }
                               }

                           }

                        //Wait for threads to finish
```

```
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thAllChp[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }


                        //Threads for executing philosophers
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim!=true)         // check for
starvation
                            {  thExPh[j]=new Threads(p[j], null, null,
'E',ruleChoice);  }
                        }

                        //Wait for thread to finish
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thExPh[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }


                        //Check for deadlock
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim==true)
                            {  DEADLOCK=false; break; }
                            else
                            {  DEADLOCK=true;  }
                        }
                        if(DEADLOCK==true)
                        {   deadlock_image.setVisible(true);  comment(3);}

                        //Check for loop
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim==false)
                            {  loop=true; break; }
                            else
                            {  loop=false;  }
                        }


                        }while(loop==true && DEADLOCK==false);

                        if(DEADLOCK==false)
                        {  stable_image.setVisible(true);  comment(4); }

                      }break;
```

```
            case 2: {  System.out.println("Rule Two");    comment(6);/// eat
if both chopstick are available

                        noOfPh=5;                       // Total Philosophers

                        thAllChp=new Thread[noOfPh];
                        thExPh=new Thread[noOfPh];


                        //set status to waiting
                        for(int i=0;i<noOfPh;i++)
                        {  p[i].upgradePhStatus();        }


                        do
                        {

                       //Threads for allocating chopsticks
                           //Allocate both chopstick
                           for(int j=0;j<noOfPh;j++)
                           {
                              if(p[j].victim!=true)     // check for
startvation
                              {
                                 switch(p[j].getPhLabel())
                                 {
                                     case 1: thAllChp[j]=new
Threads(p[j],c2,c1,'A',ruleChoice); break;
                                     case 2: thAllChp[j]=new
Threads(p[j],c3,c2,'A',ruleChoice); break;
                                     case 3: thAllChp[j]=new
Threads(p[j],c4,c3,'A',ruleChoice); break;
                                     case 4: thAllChp[j]=new
Threads(p[j],c5,c4,'A',ruleChoice); break;
                                     case 5: thAllChp[j]=new
Threads(p[j],c1,c5,'A',ruleChoice); break;

                                 }
                                 try{
                                       thAllChp[j].sleep(1000);
                                       thAllChp[j].join();
                                 }catch(InterruptedException e)
                                 { System.out.println("Error in Sleep()"); }
                               }
                              try
                              {
                                thAllChp[j].join();
                              }catch(InterruptedException e)
                               { System.out.println("Main Interrupted!");  }

                           }

                        //Wait for threads to finish
```

```
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thAllChp[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }


                        //Threads for executing philosophers
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim!=true)         // check for
starvation
                            {  thExPh[j]=new Threads(p[j], null, null,
'E',ruleChoice);  }
                        }

                        //Wait for thread to finish
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thExPh[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }


                        //Check for deadlock
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim==true)
                            {  DEADLOCK=false; break; }
                            else
                            {  DEADLOCK=true;  }
                        }
                        if(DEADLOCK==true)
                        {   deadlock_image.setVisible(true);   comment(3); }


                        //Check for loop
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim==false)
                            {  loop=true; break; }
                            else
                            {  loop=false;  }
                        }


                        }while(loop==true && DEADLOCK==false);

                        if(DEADLOCK==false)
                        {  stable_image.setVisible(true); comment(4); }

                    }break;
```

```
            case 3: {  comment(6);// Asymmetric Soln

                       noOfPh=5;                        // Total Philosophers

                       thAllChp=new Thread[noOfPh];
                       thExPh=new Thread[noOfPh];

                       //set status to waiting
                       for(int i=0;i<noOfPh;i++)
                       {  p[i].upgradePhStatus();        }


                       do
                       {

                      //Threads for allocating chopsticks

                          //Allocate first chopstick
                          for(int j=0;j<noOfPh;j++)
                          {
                             if(p[j].victim!=true)     // check for starvation
                             {
                                switch(p[j].getPhLabel())
                                {
                                    case 1: thAllChp[j]=new
Threads(p[j],c2,c1,'A',ruleChoice); break;
                                    case 2: thAllChp[j]=new
Threads(p[j],c3,c2,'A',ruleChoice); break;
                                    case 3: thAllChp[j]=new
Threads(p[j],c4,c3,'A',ruleChoice); break;
                                    case 4: thAllChp[j]=new
Threads(p[j],c5,c4,'A',ruleChoice); break;
                                    case 5: thAllChp[j]=new
Threads(p[j],c1,c5,'A',ruleChoice); break;

                                }
                                try{
                                      thAllChp[j].sleep(1000);
                                      thAllChp[j].join();
                                }catch(InterruptedException e)
                                { System.out.println("Error in Sleep()"); }
                              }

                          }

                       //Wait for threads to finish
                       try
                       {
                            for(int j=0;j<noOfPh;j++)
                            {  thAllChp[j].join();    }
                       }catch(InterruptedException e)
                       { System.out.println("Main Interrupted!");  }
```

```
                        //Allocate Second chopstick
                           for(int j=0;j<noOfPh;j++)
                           {
                              if(p[j].victim!=true)     // check for
startvation
                              {
                                 switch(p[j].getPhLabel())
                                 {
                                     case 1: thAllChp[j]=new
Threads(p[j],c2,c1,'A',ruleChoice); break;
                                     case 2: thAllChp[j]=new
Threads(p[j],c3,c2,'A',ruleChoice); break;
                                     case 3: thAllChp[j]=new
Threads(p[j],c4,c3,'A',ruleChoice); break;
                                     case 4: thAllChp[j]=new
Threads(p[j],c5,c4,'A',ruleChoice); break;
                                     case 5: thAllChp[j]=new
Threads(p[j],c1,c5,'A',ruleChoice); break;

                                 }
                                 try{
                                       thAllChp[j].sleep(1000);
                                       thAllChp[j].join();
                                 }catch(InterruptedException e)
                                 { System.out.println("Error in Sleep()"); }

                               }

                           }

                        //Wait for threads to finish
                        try
                        {
                             for(int j=0;j<noOfPh;j++)
                             {  thAllChp[j].join();    }
                        }catch(InterruptedException e)
                        { System.out.println("Main Interrupted!");  }




                        //Threads for executing philosophers
                        for(int j=0;j<noOfPh;j++)
                        {
                            if(p[j].victim!=true)         // check for
startvation
                            {  thExPh[j]=new Threads(p[j], null, null,
'E',ruleChoice);  }
                        }

                        //Wait for thread to finish
                        try
                        {
```

```
                        for(int j=0;j<noOfPh;j++)
                        {  thExPh[j].join();    }
                    }catch(InterruptedException e)
                    { System.out.println("Main Interrupted!");  }


                    //Check for deadlock
                    for(int j=0;j<noOfPh;j++)
                    {
                        if(p[j].victim==true)
                        {  DEADLOCK=false; break; }
                        else
                        {  DEADLOCK=true;  }
                    }
                    if(DEADLOCK==true)
                    {   deadlock_image.setVisible(true);  comment(3); }

                    //Check for loop
                    for(int j=0;j<noOfPh;j++)
                    {
                        if(p[j].victim==false)
                        {  loop=true; break; }
                        else
                        {  loop=false;  }
                    }


                    }while(loop==true && DEADLOCK==false);

                    if(DEADLOCK==false)
                    {  stable_image.setVisible(true); comment(4); }

                }break;

        }//end of switch

        if(ruleChoice!=-1)
        {    start_deactive.setVisible(false);
             reset_button.setVisible(true);
        }

    }

    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">//GEN-
BEGIN:initComponents
    private void initComponents() {

        buttonGroup_rule = new javax.swing.ButtonGroup();
        cmt_welcom = new javax.swing.JLabel();
        cmt_select = new javax.swing.JLabel();
        cmt_deadlock = new javax.swing.JLabel();
        cmt_stable = new javax.swing.JLabel();
        cmt_optSel = new javax.swing.JLabel();
```

```
cmt_optRun = new javax.swing.JLabel();
cmt_moreInfo = new javax.swing.JLabel();
ph1_eat = new javax.swing.JLabel();
ph1_thinking = new javax.swing.JLabel();
ph1_wt_0 = new javax.swing.JLabel();
ph1_wt_L = new javax.swing.JLabel();
ph1_wt_R = new javax.swing.JLabel();
ph1_timmer = new javax.swing.JLabel();
ph2_eat = new javax.swing.JLabel();
ph2_thinking = new javax.swing.JLabel();
ph2_wt_0 = new javax.swing.JLabel();
ph2_wt_L = new javax.swing.JLabel();
ph2_wt_R = new javax.swing.JLabel();
ph2_timmer = new javax.swing.JLabel();
ph3_eat = new javax.swing.JLabel();
ph3_thinking = new javax.swing.JLabel();
ph3_wt_0 = new javax.swing.JLabel();
ph3_wt_L = new javax.swing.JLabel();
ph3_wt_R = new javax.swing.JLabel();
ph3_timmer = new javax.swing.JLabel();
ph4_eat = new javax.swing.JLabel();
ph4_thinking = new javax.swing.JLabel();
ph4_wt_0 = new javax.swing.JLabel();
ph4_wt_L = new javax.swing.JLabel();
ph4_wt_R = new javax.swing.JLabel();
ph4_timmer = new javax.swing.JLabel();
ph5_eat = new javax.swing.JLabel();
ph5_thinking = new javax.swing.JLabel();
ph5_wt_0 = new javax.swing.JLabel();
ph5_wt_L = new javax.swing.JLabel();
ph5_wt_R = new javax.swing.JLabel();
ph5_timmer = new javax.swing.JLabel();
sad_ph5 = new javax.swing.JLabel();
chp1_image = new javax.swing.JLabel();
chp2_image = new javax.swing.JLabel();
chp3_image = new javax.swing.JLabel();
chp4_image = new javax.swing.JLabel();
chp5_image = new javax.swing.JLabel();
deadlock_image = new javax.swing.JLabel();
stable_image = new javax.swing.JLabel();
rdBtn_0 = new javax.swing.JRadioButton();
rdBtn_1 = new javax.swing.JRadioButton();
rdBtn_2 = new javax.swing.JRadioButton();
rdBtn_3 = new javax.swing.JRadioButton();
prbStmt_button = new javax.swing.JButton();
about_button = new javax.swing.JButton();
start_button = new javax.swing.JButton();
reset_button = new javax.swing.JButton();
start_deactive = new javax.swing.JLabel();
screen2_bg = new javax.swing.JLabel();

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
setTitle("The Dining Philosopher Problem | (c) 2011Gagandeep Singh");
setAlwaysOnTop(true);
```

```
        setBounds(new java.awt.Rectangle(0, 0, 900, 700));
        setMinimumSize(new java.awt.Dimension(900, 700));
        setResizable(false);
        getContentPane().setLayout(null);

        cmt_welcom.setFont(new java.awt.Font("Comic Sans MS", 0, 16));
        cmt_welcom.setText("Welcome to Dining Philosopher Problem......");
        getContentPane().add(cmt_welcom);
        cmt_welcom.setBounds(180, 597, 360, 30);

        cmt_select.setFont(new java.awt.Font("Comic Sans MS", 0, 15));
        cmt_select.setText("Please Select an option and click 'Start'
Button!");
        getContentPane().add(cmt_select);
        cmt_select.setBounds(180, 617, 380, 30);

        cmt_deadlock.setFont(new java.awt.Font("Comic Sans MS", 0, 16));
        cmt_deadlock.setText("Deadlock occured !");
        getContentPane().add(cmt_deadlock);
        cmt_deadlock.setBounds(180, 593, 230, 40);

        cmt_stable.setFont(new java.awt.Font("Comic Sans MS", 0, 16));
        cmt_stable.setText("System is Stable !");
        getContentPane().add(cmt_stable);
        cmt_stable.setBounds(180, 595, 150, 40);

        cmt_optSel.setFont(new java.awt.Font("Comic Sans MS", 0, 16));
        cmt_optSel.setText("Option Selected!");
        getContentPane().add(cmt_optSel);
        cmt_optSel.setBounds(180, 593, 330, 50);

        cmt_optRun.setFont(new java.awt.Font("Comic Sans MS", 0, 16));
        cmt_optRun.setText("Running......");
        getContentPane().add(cmt_optRun);
        cmt_optRun.setBounds(180, 593, 310, 50);

        cmt_moreInfo.setFont(new java.awt.Font("Arial", 0, 11));
        cmt_moreInfo.setText("For More Information click 'Problem
Statement'.");
        getContentPane().add(cmt_moreInfo);
        cmt_moreInfo.setBounds(237, 646, 230, 14);

        ph1_eat.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph1_eating.gi
f"))); // NOI18N
        getContentPane().add(ph1_eat);
        ph1_eat.setBounds(272, 87, 180, 130);

        ph1_thinking.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph1_thinking.
gif"))); // NOI18N
        getContentPane().add(ph1_thinking);
        ph1_thinking.setBounds(260, 79, 241, 124);
```

```
        ph1_wt_0.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph1_wt_0.gif"
))); // NOI18N
        getContentPane().add(ph1_wt_0);
        ph1_wt_0.setBounds(260, 60, 250, 160);

        ph1_wt_L.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph1_wt_L.gif"
))); // NOI18N
        getContentPane().add(ph1_wt_L);
        ph1_wt_L.setBounds(261, 76, 250, 130);

        ph1_wt_R.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph1_wt_R.gif"
))); // NOI18N
        getContentPane().add(ph1_wt_R);
        ph1_wt_R.setBounds(261, 66, 240, 150);

        ph1_timmer.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Timmer.gif"))
); // NOI18N
        getContentPane().add(ph1_timmer);
        ph1_timmer.setBounds(272, 75, 140, 30);

        ph2_eat.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph2_eating.gi
f"))); // NOI18N
        getContentPane().add(ph2_eat);
        ph2_eat.setBounds(447, 218, 150, 140);

        ph2_thinking.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph2_thinking.
gif"))); // NOI18N
        getContentPane().add(ph2_thinking);
        ph2_thinking.setBounds(439, 205, 250, 150);

        ph2_wt_0.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph2_wt_0.gif"
))); // NOI18N
        getContentPane().add(ph2_wt_0);
        ph2_wt_0.setBounds(440, 210, 250, 140);

        ph2_wt_L.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph2_wt_L.gif"
))); // NOI18N
        getContentPane().add(ph2_wt_L);
        ph2_wt_L.setBounds(440, 205, 250, 150);

        ph2_wt_R.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph2_wt_R.gif"
))); // NOI18N
        ph2_wt_R.setText("jLabel1");
        getContentPane().add(ph2_wt_R);
        ph2_wt_R.setBounds(440, 205, 250, 150);
```

```
        ph2_timmer.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Timmer.gif"))
); // NOI18N
        getContentPane().add(ph2_timmer);
        ph2_timmer.setBounds(530, 210, 140, 30);

        ph3_eat.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph3_eating.gi
f"))); // NOI18N
        getContentPane().add(ph3_eat);
        ph3_eat.setBounds(368, 410, 180, 130);

        ph3_thinking.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph3_thinking.
gif"))); // NOI18N
        getContentPane().add(ph3_thinking);
        ph3_thinking.setBounds(360, 390, 270, 160);

        ph3_wt_0.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph3_wt_0.gif"
))); // NOI18N
        getContentPane().add(ph3_wt_0);
        ph3_wt_0.setBounds(360, 400, 270, 140);

        ph3_wt_L.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph3_wt_L.gif"
))); // NOI18N
        getContentPane().add(ph3_wt_L);
        ph3_wt_L.setBounds(360, 400, 270, 140);

        ph3_wt_R.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph3_wt_R.gif"
))); // NOI18N
        getContentPane().add(ph3_wt_R);
        ph3_wt_R.setBounds(360, 405, 270, 130);

        ph3_timmer.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Timmer.gif"))
); // NOI18N
        getContentPane().add(ph3_timmer);
        ph3_timmer.setBounds(490, 510, 140, 30);

        ph4_eat.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph4_eating.gi
f"))); // NOI18N
        getContentPane().add(ph4_eat);
        ph4_eat.setBounds(169, 405, 140, 130);

        ph4_thinking.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph4_thinking.
gif"))); // NOI18N
        getContentPane().add(ph4_thinking);
        ph4_thinking.setBounds(39, 406, 260, 130);
```

```
        ph4_wt_0.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph4_wt_0.gif"
))); // NOI18N
        getContentPane().add(ph4_wt_0);
        ph4_wt_0.setBounds(39, 406, 260, 130);

        ph4_wt_L.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph4_wt_L.gif"
))); // NOI18N
        getContentPane().add(ph4_wt_L);
        ph4_wt_L.setBounds(39, 397, 270, 150);

        ph4_wt_R.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph4_wt_R.gif"
))); // NOI18N
        getContentPane().add(ph4_wt_R);
        ph4_wt_R.setBounds(39, 401, 270, 140);

        ph4_timmer.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Timmer.gif"))
); // NOI18N
        getContentPane().add(ph4_timmer);
        ph4_timmer.setBounds(160, 520, 140, 30);

        ph5_eat.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph5_eating.gi
f"))); // NOI18N
        getContentPane().add(ph5_eat);
        ph5_eat.setBounds(104, 222, 150, 130);

        ph5_thinking.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph5_thinking.
gif"))); // NOI18N
        getContentPane().add(ph5_thinking);
        ph5_thinking.setBounds(16, 176, 240, 190);

        ph5_wt_0.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph5_wt_0.gif"
))); // NOI18N
        getContentPane().add(ph5_wt_0);
        ph5_wt_0.setBounds(16, 178, 220, 187);

        ph5_wt_L.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph5_wt_L.gif"
))); // NOI18N
        getContentPane().add(ph5_wt_L);
        ph5_wt_L.setBounds(16, 176, 230, 190);

        ph5_wt_R.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/ph5_wt_R.gif"
))); // NOI18N
        getContentPane().add(ph5_wt_R);
        ph5_wt_R.setBounds(16, 176, 230, 190);
```

```
        ph5_timmer.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Timmer.gif"))
); // NOI18N
        getContentPane().add(ph5_timmer);
        ph5_timmer.setBounds(20, 190, 140, 30);

        sad_ph5.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Sad_ph5.gif")
)); // NOI18N
        getContentPane().add(sad_ph5);
        sad_ph5.setBounds(10, 130, 130, 160);

        chp1_image.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/chopstick1.gi
f"))); // NOI18N
        getContentPane().add(chp1_image);
        chp1_image.setBounds(262, 229, 60, 70);

        chp2_image.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/chopstick2.gi
f"))); // NOI18N
        getContentPane().add(chp2_image);
        chp2_image.setBounds(369, 237, 43, 59);

        chp3_image.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/chopstick3.gi
f"))); // NOI18N
        getContentPane().add(chp3_image);
        chp3_image.setBounds(385, 343, 80, 40);

        chp4_image.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/chopstick4.gi
f"))); // NOI18N
        getContentPane().add(chp4_image);
        chp4_image.setBounds(332, 380, 20, 80);

        chp5_image.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/chopstick5.gi
f"))); // NOI18N
        getContentPane().add(chp5_image);
        chp5_image.setBounds(218, 342, 70, 40);

        deadlock_image.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Deadlock_2.gi
f"))); // NOI18N
        getContentPane().add(deadlock_image);
        deadlock_image.setBounds(697, 555, 190, 80);

        stable_image.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Stable_2.gif"
))); // NOI18N
        getContentPane().add(stable_image);
        stable_image.setBounds(698, 606, 190, 70);
```

```
        buttonGroup_rule.add(rdBtn_0);
        rdBtn_0.setBorder(null);
        rdBtn_0.setContentAreaFilled(false);
        rdBtn_0.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        rdBtn_0.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
1.gif"))); // NOI18N
        rdBtn_0.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
2.gif"))); // NOI18N
        rdBtn_0.setRolloverSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
4.gif"))); // NOI18N
        rdBtn_0.setSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
3.gif"))); // NOI18N
        rdBtn_0.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                rdBtn_0ActionPerformed(evt);
            }
        });
        getContentPane().add(rdBtn_0);
        rdBtn_0.setBounds(700, 250, 20, 19);

        buttonGroup_rule.add(rdBtn_1);
        rdBtn_1.setContentAreaFilled(false);
        rdBtn_1.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        rdBtn_1.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
1.gif"))); // NOI18N
        rdBtn_1.setMaximumSize(new java.awt.Dimension(19, 19));
        rdBtn_1.setMinimumSize(new java.awt.Dimension(19, 19));
        rdBtn_1.setPreferredSize(new java.awt.Dimension(19, 19));
        rdBtn_1.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
2.gif"))); // NOI18N
        rdBtn_1.setRolloverSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
4.gif"))); // NOI18N
        rdBtn_1.setSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
3.gif"))); // NOI18N
        rdBtn_1.setVerifyInputWhenFocusTarget(false);
        rdBtn_1.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                rdBtn_1ActionPerformed(evt);
            }
        });
        getContentPane().add(rdBtn_1);
        rdBtn_1.setBounds(695, 340, 30, 20);

        buttonGroup_rule.add(rdBtn_2);
        rdBtn_2.setContentAreaFilled(false);
```

```
        rdBtn_2.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        rdBtn_2.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
1.gif"))); // NOI18N
        rdBtn_2.setMaximumSize(new java.awt.Dimension(19, 19));
        rdBtn_2.setMinimumSize(new java.awt.Dimension(19, 19));
        rdBtn_2.setPreferredSize(new java.awt.Dimension(19, 19));
        rdBtn_2.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
2.gif"))); // NOI18N
        rdBtn_2.setRolloverSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
4.gif"))); // NOI18N
        rdBtn_2.setSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
3.gif"))); // NOI18N
        rdBtn_2.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                rdBtn_2ActionPerformed(evt);
            }
        });
        getContentPane().add(rdBtn_2);
        rdBtn_2.setBounds(695, 388, 30, 19);

        buttonGroup_rule.add(rdBtn_3);
        rdBtn_3.setContentAreaFilled(false);
        rdBtn_3.setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        rdBtn_3.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
1.gif"))); // NOI18N
        rdBtn_3.setMaximumSize(new java.awt.Dimension(19, 19));
        rdBtn_3.setMinimumSize(new java.awt.Dimension(19, 19));
        rdBtn_3.setPreferredSize(new java.awt.Dimension(19, 19));
        rdBtn_3.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
2.gif"))); // NOI18N
        rdBtn_3.setRolloverSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
4.gif"))); // NOI18N
        rdBtn_3.setSelectedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/Radio_button_
3.gif"))); // NOI18N
        rdBtn_3.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                rdBtn_3ActionPerformed(evt);
            }
        });
        getContentPane().add(rdBtn_3);
        rdBtn_3.setBounds(695, 434, 40, 19);

        prbStmt_button.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/PrbSt_button_
1.gif"))); // NOI18N
        prbStmt_button.setBorder(null);
```

```
        prbStmt_button.setBorderPainted(false);
        prbStmt_button.setContentAreaFilled(false);
        prbStmt_button.setCursor(new
java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        prbStmt_button.setPressedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/PrbSt_button_
3.gif"))); // NOI18N
        prbStmt_button.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/PrbSt_button_
2.gif"))); // NOI18N
        prbStmt_button.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                prbStmt_buttonActionPerformed(evt);
            }
        });
        getContentPane().add(prbStmt_button);
        prbStmt_button.setBounds(730, 140, 130, 40);

        about_button.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/About_button_
1.gif"))); // NOI18N
        about_button.setBorder(null);
        about_button.setBorderPainted(false);
        about_button.setContentAreaFilled(false);
        about_button.setCursor(new
java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        about_button.setPressedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/About_button_
3.gif"))); // NOI18N
        about_button.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/About_button_
2.gif"))); // NOI18N
        about_button.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                about_buttonActionPerformed(evt);
            }
        });
        getContentPane().add(about_button);
        about_button.setBounds(730, 100, 130, 40);

        start_button.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/start_button_
1.gif"))); // NOI18N
        start_button.setBorder(null);
        start_button.setBorderPainted(false);
        start_button.setContentAreaFilled(false);
        start_button.setCursor(new
java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        start_button.setFocusPainted(false);
        start_button.setPressedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/start_button_
4.gif"))); // NOI18N
```

```
        start_button.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/start_button_
2.gif"))); // NOI18N
        start_button.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                start_buttonActionPerformed(evt);
            }
        });
        getContentPane().add(start_button);
        start_button.setBounds(540, 550, 150, 120);

        reset_button.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/reset_button_
1.gif"))); // NOI18N
        reset_button.setBorderPainted(false);
        reset_button.setContentAreaFilled(false);
        reset_button.setCursor(new
java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
        reset_button.setPressedIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/reset_button_
3.gif"))); // NOI18N
        reset_button.setRolloverIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/reset_button_
2.gif"))); // NOI18N
        reset_button.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                reset_buttonActionPerformed(evt);
            }
        });
        getContentPane().add(reset_button);
        reset_button.setBounds(550, 550, 130, 110);

        start_deactive.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/start_button_
4.gif"))); // NOI18N
        getContentPane().add(start_deactive);
        start_deactive.setBounds(546, 550, 150, 120);

        screen2_bg.setIcon(new
javax.swing.ImageIcon(getClass().getResource("/diningphilosopher/screen2_bg.jp
g"))); // NOI18N
        getContentPane().add(screen2_bg);
        screen2_bg.setBounds(0, 0, 900, 675);

        pack();
    }// </editor-fold>//GEN-END:initComponents

    private void about_buttonActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_about_buttonActionPerformed
        new AboutScreen().setVisible(true);
    }//GEN-LAST:event_about_buttonActionPerformed

    private void prbStmt_buttonActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_prbStmt_buttonActionPerformed
```

```
        // TODO add your handling code here:
        new PrbStScreen().setVisible(true);
    }//GEN-LAST:event_prbStmt_buttonActionPerformed

    private void rdBtn_1ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_rdBtn_1ActionPerformed
        ruleChoice=1;
        comment(5);
    }//GEN-LAST:event_rdBtn_1ActionPerformed

    //Start Button
    private void start_buttonActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_start_buttonActionPerformed

        if(ruleChoice!=-1)
        {
            exe=true;
            System.out.println("Value of 'exe' :"+exe);
            start_button.setVisible(false);
            start_deactive.setVisible(true);
        }


    }//GEN-LAST:event_start_buttonActionPerformed

    private void rdBtn_0ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_rdBtn_0ActionPerformed
        ruleChoice=0;
        comment(5);
    }//GEN-LAST:event_rdBtn_0ActionPerformed

    private void rdBtn_2ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_rdBtn_2ActionPerformed
        ruleChoice=2;
        comment(5);
    }//GEN-LAST:event_rdBtn_2ActionPerformed

    private void rdBtn_3ActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_rdBtn_3ActionPerformed
        ruleChoice=3;
        comment(5);
    }//GEN-LAST:event_rdBtn_3ActionPerformed

    private void reset_buttonActionPerformed(java.awt.event.ActionEvent evt)
{//GEN-FIRST:event_reset_buttonActionPerformed

        for(int i=0;i<noOfPh;i++)
        {  p[i].reset();   }

        c1.reset();
        c2.reset();
        c3.reset();
        c4.reset();
        c5.reset();
```

```
    deadlock_image.setVisible(false);
    stable_image.setVisible(false);
    start_button.setVisible(true);
    reset_button.setVisible(false);
    ph5_thinking.setVisible(true);
    sad_ph5.setVisible(false);

    start_button.setVisible(true);
    start_deactive.setVisible(false);

    comment(2);
}//GEN-LAST:event_reset_buttonActionPerformed



public static void main(String args[])
{
    java.awt.EventQueue.invokeLater(new Runnable()
    {
        public void run()
        {
            new Screen1().setVisible(true);
            //new Screen2().setVisible(true);
        }
    });

    do
    {
        if(exe==true)
        {  System.out.println("Running algo....");
           EXECUTEALGO();
           exe=false;
        }

    }while(true);

}

private static void hideAllComments()
{

    cmt_welcom.setVisible(false);
    cmt_select.setVisible(false);
    cmt_deadlock.setVisible(false);
    cmt_stable.setVisible(false);
    cmt_optSel.setVisible(false);
    cmt_optRun.setVisible(false);
}

public static void comment(int c)
{
    hideAllComments();
```

```
    switch(c)
    {
        case 1: cmt_welcom.setVisible(true); break;
        case 2: cmt_select.setVisible(true); break;
        case 3: cmt_deadlock.setVisible(true); break;
        case 4: cmt_stable.setVisible(true); break;
        case 5: cmt_optSel.setVisible(true); break;
        case 6: cmt_optRun.setVisible(true); break;
    }

}


// Variables declaration
private javax.swing.JButton about_button;
private javax.swing.ButtonGroup buttonGroup_rule;
public javax.swing.JLabel chp1_image;
private javax.swing.JLabel chp2_image;
private javax.swing.JLabel chp3_image;
private javax.swing.JLabel chp4_image;
private javax.swing.JLabel chp5_image;
private static javax.swing.JLabel cmt_deadlock;
private static javax.swing.JLabel cmt_moreInfo;
private static javax.swing.JLabel cmt_optRun;
private static javax.swing.JLabel cmt_optSel;
private static javax.swing.JLabel cmt_select;
private static javax.swing.JLabel cmt_stable;
private static javax.swing.JLabel cmt_welcom;
private static javax.swing.JLabel deadlock_image;
private javax.swing.JLabel ph1_eat;
private javax.swing.JLabel ph1_thinking;
private javax.swing.JLabel ph1_timmer;
private javax.swing.JLabel ph1_wt_0;
private javax.swing.JLabel ph1_wt_L;
private javax.swing.JLabel ph1_wt_R;
private javax.swing.JLabel ph2_eat;
private javax.swing.JLabel ph2_thinking;
private javax.swing.JLabel ph2_timmer;
private javax.swing.JLabel ph2_wt_0;
private javax.swing.JLabel ph2_wt_L;
private javax.swing.JLabel ph2_wt_R;
private javax.swing.JLabel ph3_eat;
private javax.swing.JLabel ph3_thinking;
private javax.swing.JLabel ph3_timmer;
private javax.swing.JLabel ph3_wt_0;
private javax.swing.JLabel ph3_wt_L;
private javax.swing.JLabel ph3_wt_R;
private javax.swing.JLabel ph4_eat;
private javax.swing.JLabel ph4_thinking;
private javax.swing.JLabel ph4_timmer;
private javax.swing.JLabel ph4_wt_0;
private javax.swing.JLabel ph4_wt_L;
private javax.swing.JLabel ph4_wt_R;
private javax.swing.JLabel ph5_eat;
```

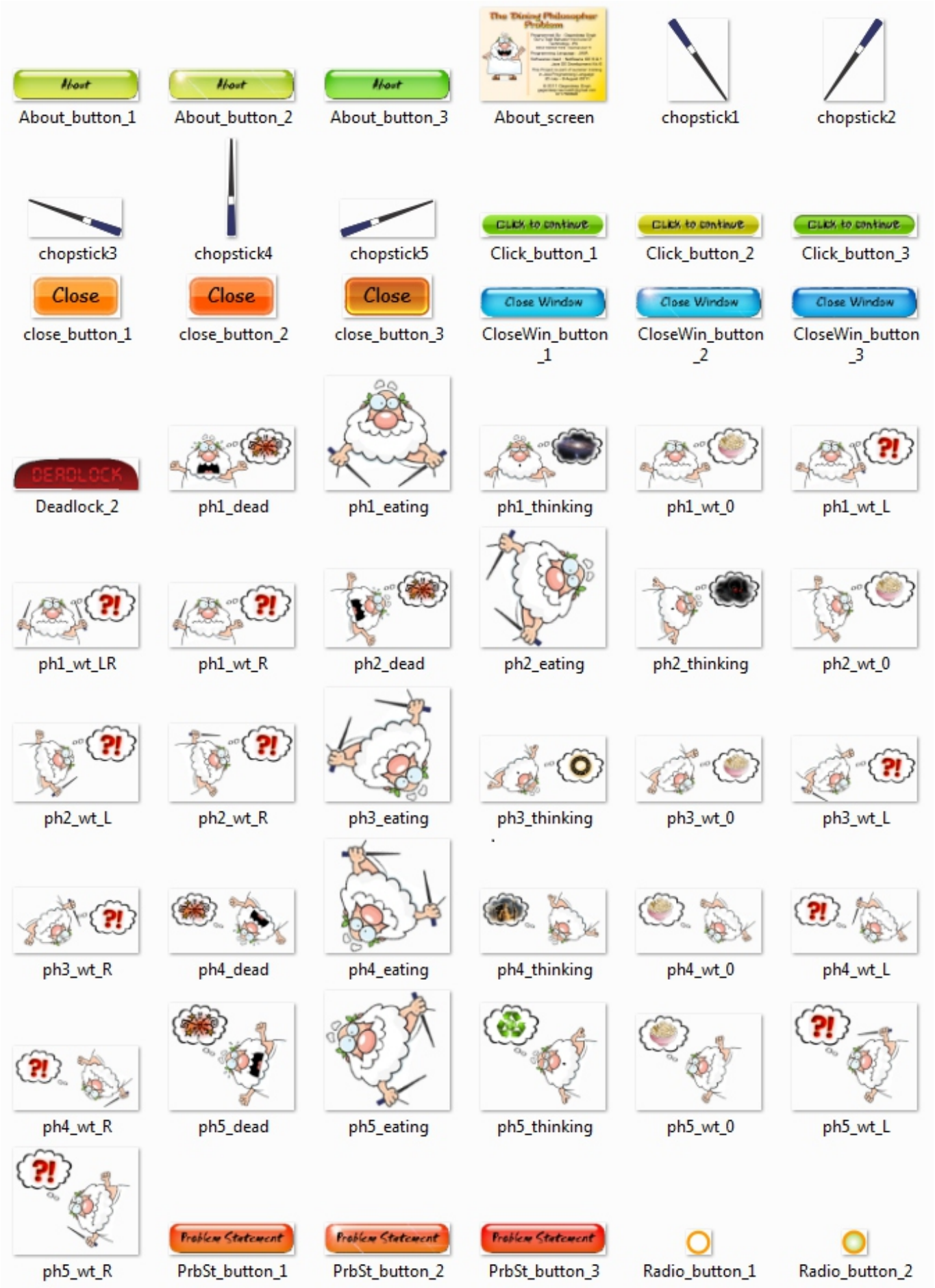
```
    private static javax.swing.JLabel ph5_thinking;
    private javax.swing.JLabel ph5_timmer;
    private javax.swing.JLabel ph5_wt_0;
    private javax.swing.JLabel ph5_wt_L;
    private javax.swing.JLabel ph5_wt_R;
    private javax.swing.JButton prbStmt_button;
    private javax.swing.JRadioButton rdBtn_0;
    private javax.swing.JRadioButton rdBtn_1;
    private javax.swing.JRadioButton rdBtn_2;
    private javax.swing.JRadioButton rdBtn_3;
    private static javax.swing.JButton reset_button;
    private static javax.swing.JLabel sad_ph5;
    private javax.swing.JLabel screen2_bg;
    private static javax.swing.JLabel stable_image;
    private static javax.swing.JButton start_button;
    private static javax.swing.JLabel start_deactive;
    // End of variables declaration

}
```

*****

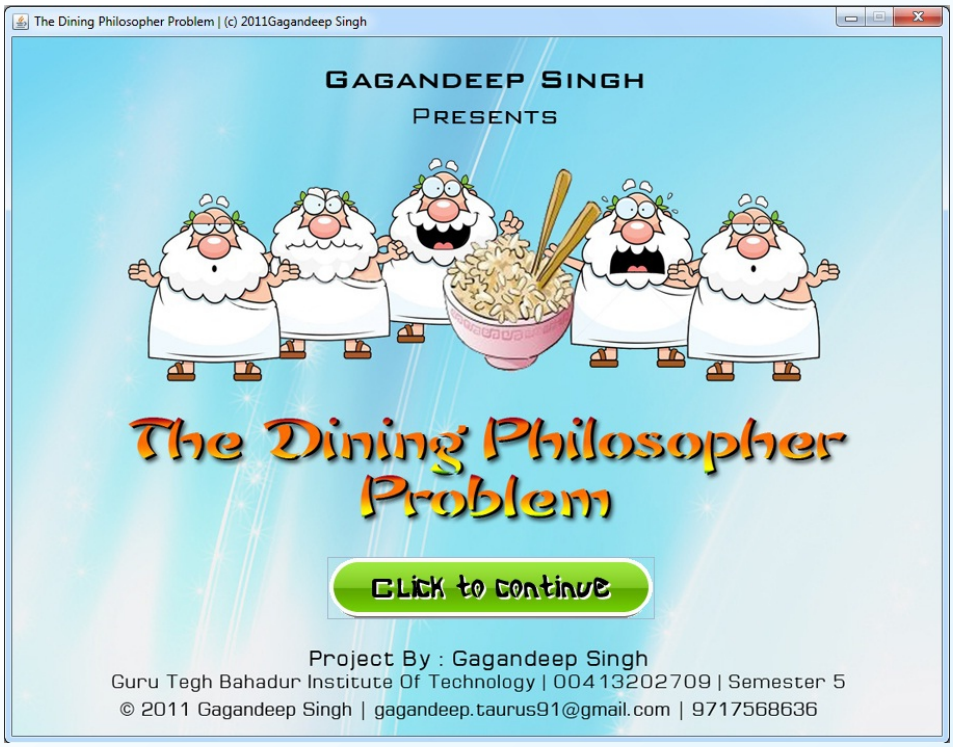
(Lines of Code : 1923 approximately)

Image files


About_button_1
About_button_2
About_button_3
About_screen
chopstick1
chopstick2
chopstick3
chopstick4
chopstick5
Click_button_1
Click_button_2
Click_button_3
close_button_1
close_button_2
close_button_3
CloseWin_button_1
CloseWin_button_2
CloseWin_button_3
Deadlock_2
ph1_dead
ph1_eating
ph1_thinking
ph1_wt_0
ph1_wt_L
ph1_wt_LR
ph1_wt_R
ph2_dead
ph2_eating
ph2_thinking
ph2_wt_0
ph2_wt_L
ph2_wt_R
ph3_eating
ph3_thinking
ph3_wt_0
ph3_wt_L
ph3_wt_R
ph4_dead
ph4_eating
ph4_thinking
ph4_wt_0
ph4_wt_L
ph4_wt_R
ph5_dead
ph5_eating
ph5_thinking
ph5_wt_0
ph5_wt_L
ph5_wt_R
PrbSt_button_1
PrbSt_button_2
PrbSt_button_3
Radio_button_1
Radio_button_2

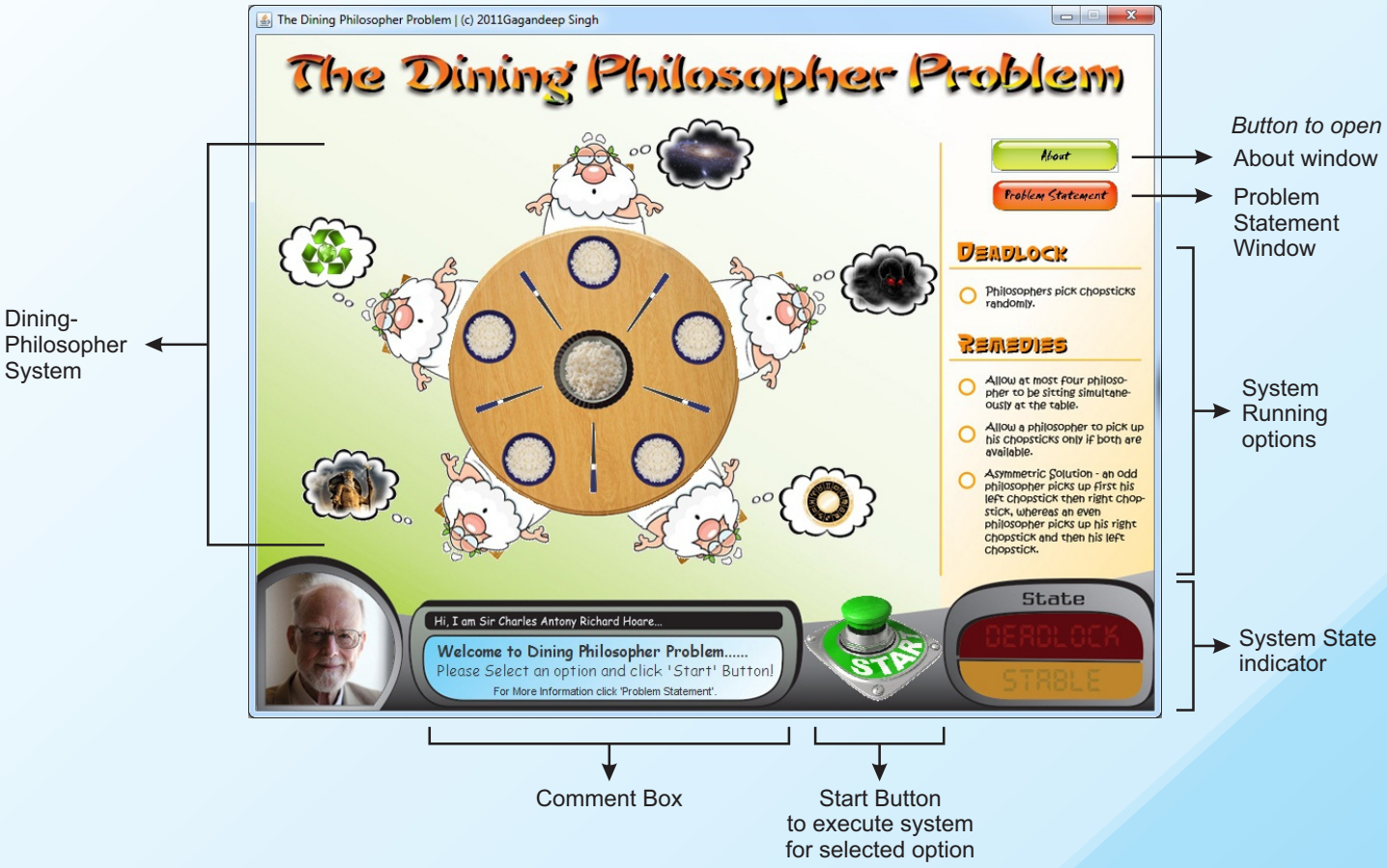

Radio_button_3
Radio_button_4
reset_button_1
reset_button_2
reset_button_3
Sad_ph5
STABLE
Stable_2
start_button_1
start_button_2
start_button_3
start_button_4
Timmer
prb_stmt_bg
screen1_bg
screen2_bg

ScreenShots



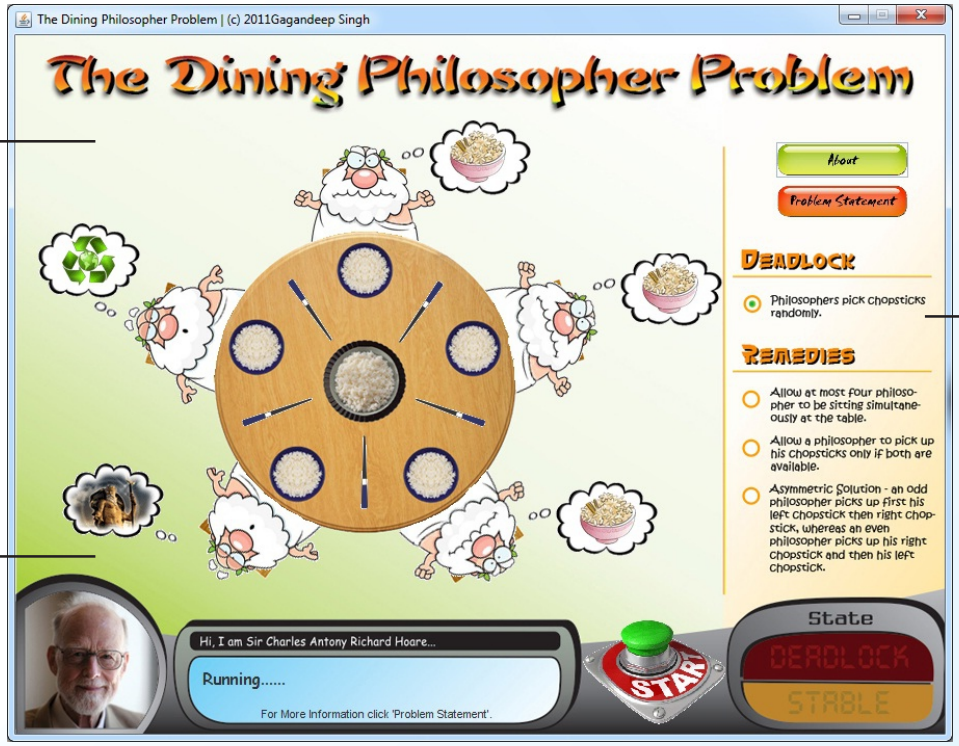
ScreenShot - 1 : Title Window

This is the first window to be displayed when the program runs. The windows displays the title of the project and contains a button to open another window describing the system.



ScreenShot - 2 : Main Window

ScreenShot - 3 : System Running

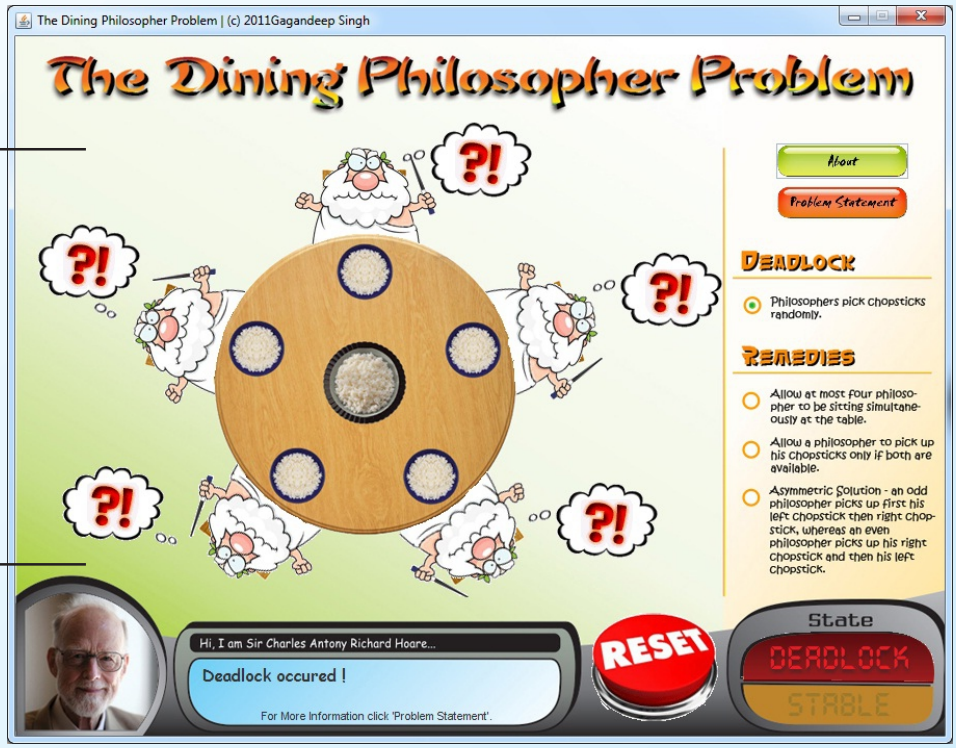


Philosophers getting hungry

Option selected to execute system for illustrating deadlock condition

Start Button Freezed

ScreenShot - 4 : Deadlock

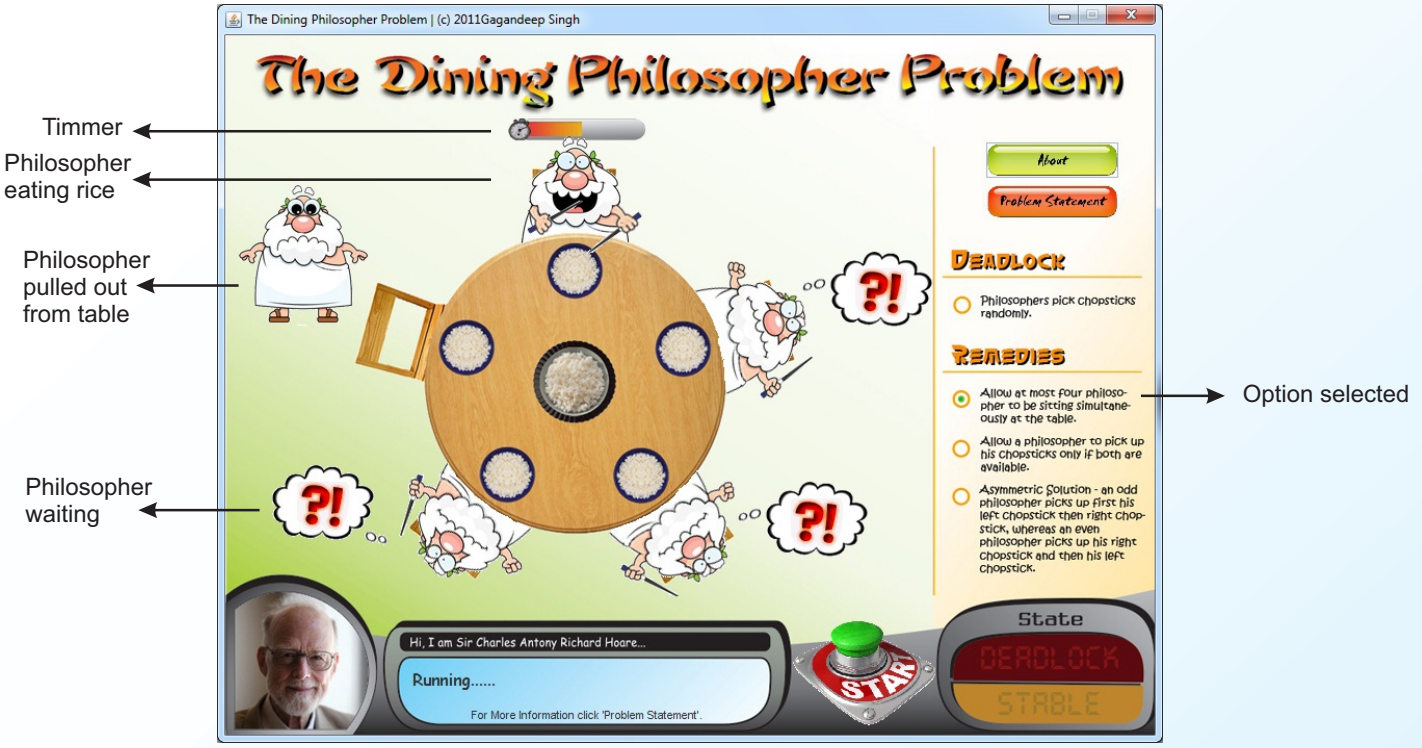


Philosophers in deadlock *Every philosopher picks up a chopstick and now wait for one another to release his own chopstick.*

Indicator indicating System in deadlock

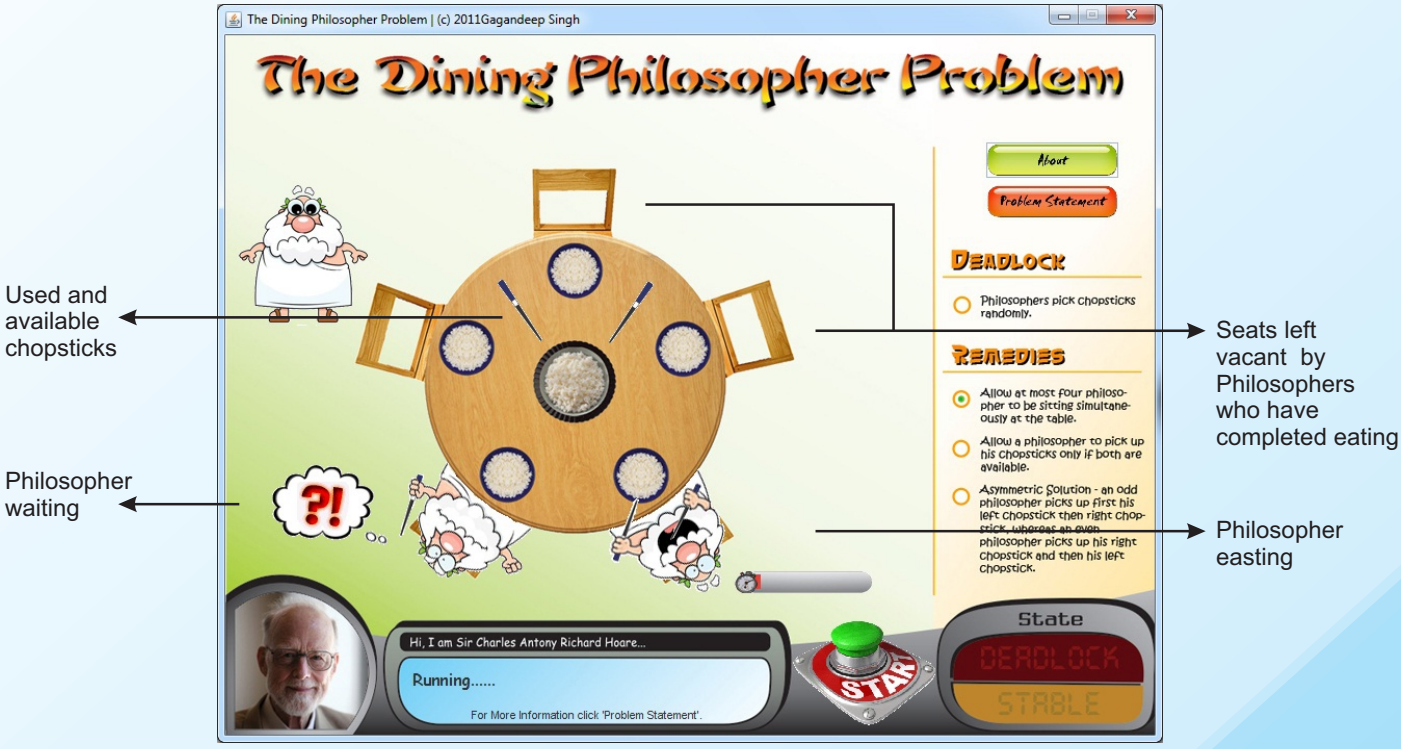
Reset Button (in place of start button) to reset system to its initial state.

ScreenShot - 5 : System Running ‘Remedy 1’ option

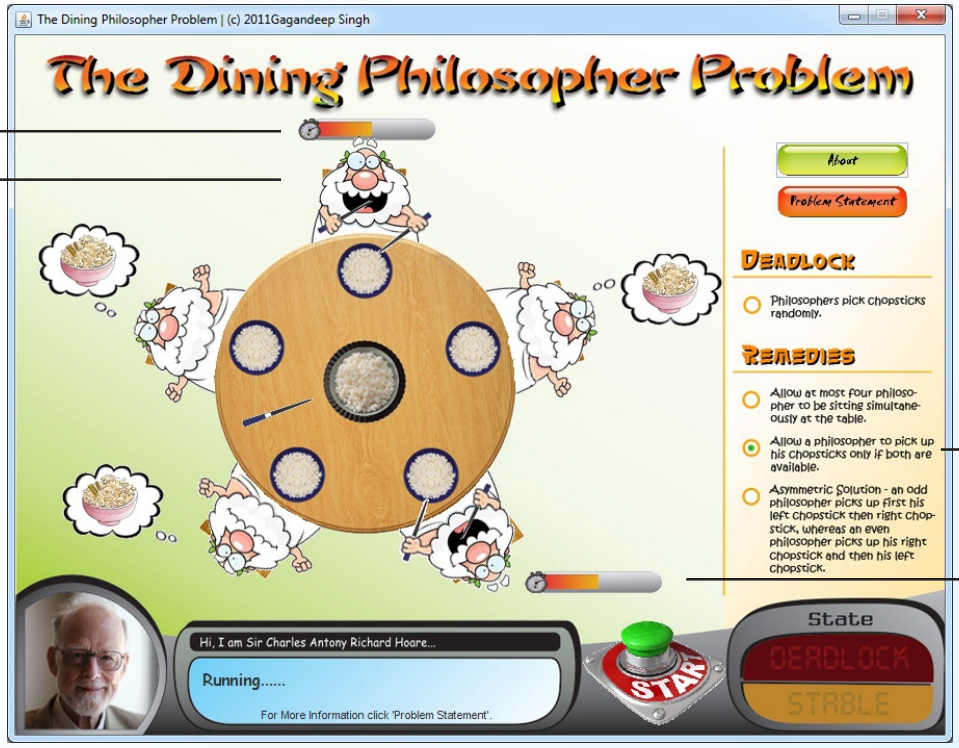


Absence of one philosopher from table makes sure that there is always one chopstick available on the table when each philosopher has got one chopstick each. Thus, the philosopher sitting adjacent to empty seat picks up the free chopstick and eats for five second. Once he is over with eating, he releases his chopsticks which are picked up by his next adjacent neighbor allowing him to eat. The system follows a chain reaction where each philosopher is able to get a chance to eat one after another.

ScreenShot - 6 : System running

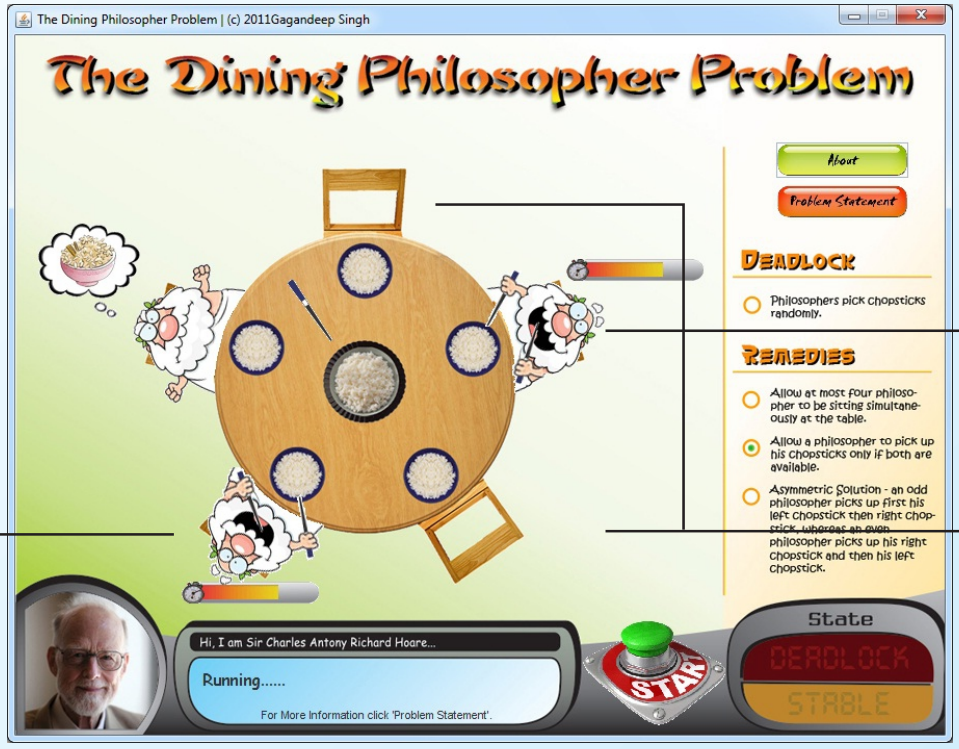


ScreenShot - 7 : System Running ‘Remedy 2’ option

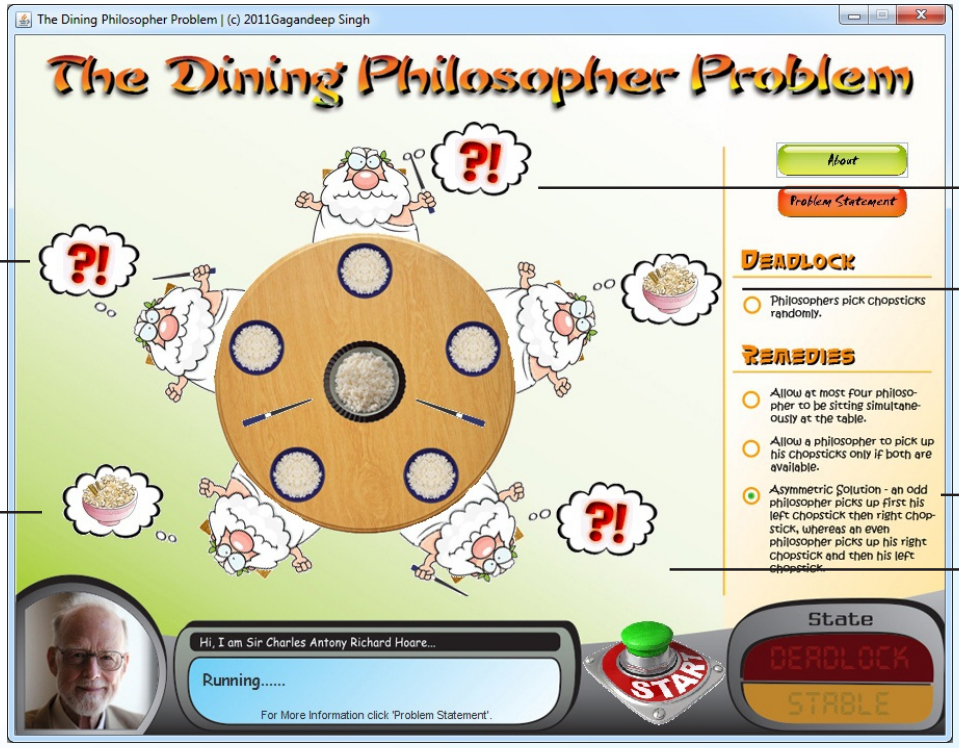


In this mode, a philosopher picks chopsticks only if both are available. In such a way two philosophers sitting in alternate position are able to acquire both the chopsticks and thus eat while other wait for their chance.

ScreenShot - 8 : System running



ScreenShot - 9 : System Running ‘Remedy 3’ option



Philosopher 5 owns left chopstick

Philosopher 4 cannot pick left chopstick

philosopher 1 picks up his left chopstick

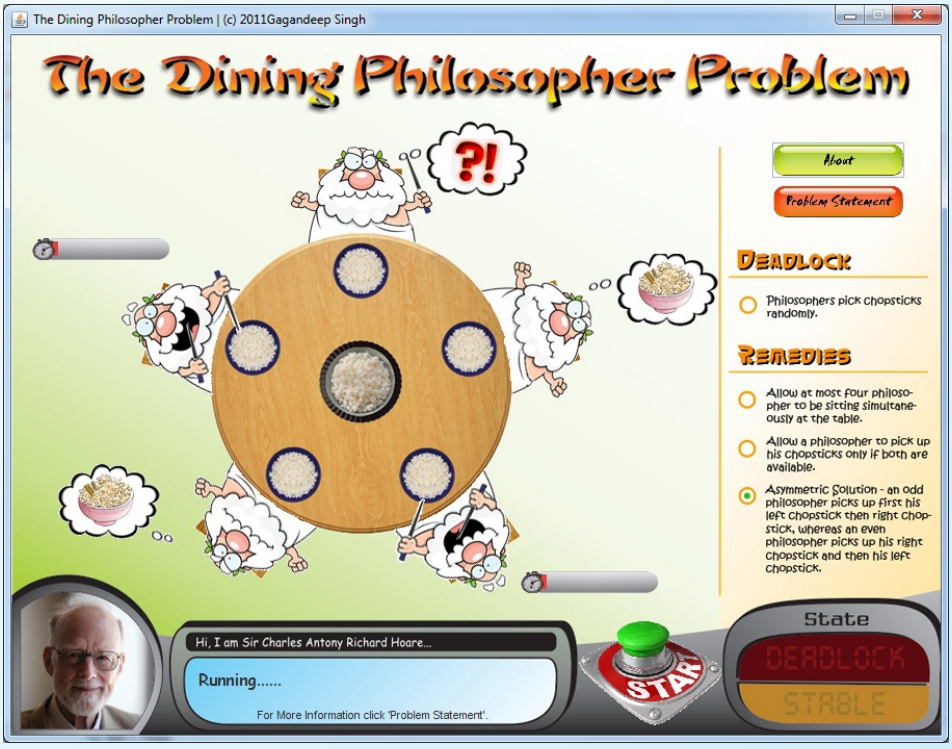
philosopher 2 cannot pick chopstick since he is allowed to pick his right chopstick first.

Option selected

Philosopher 3 gets left chopstick

In last mode, a philosopher with odd label number pick his left chopstick first and then right chopstick whereas the philosopher with even label number picks his right chopstick first and then his left chopstick.

ScreenShot - 10 : System running

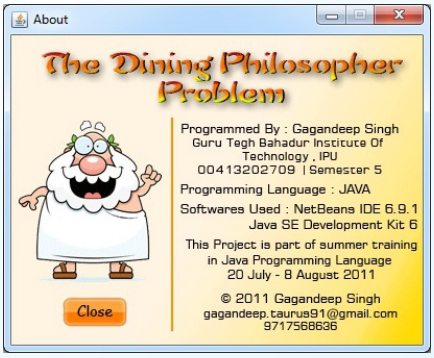


ScreenShot - 11 : System in Safe State

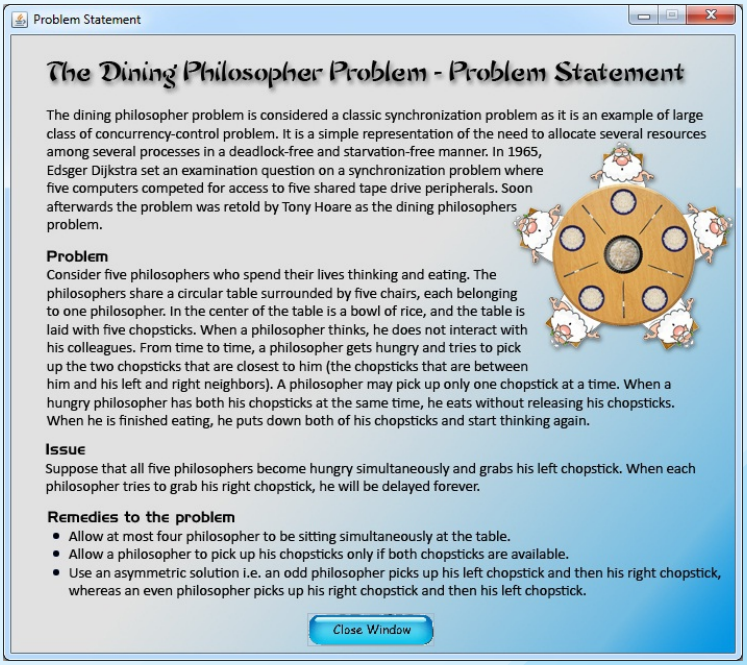
Vacant dining Table indicating all philosophers were able to eat rice.

Indicator indicating System in Safe State.

Reset Button



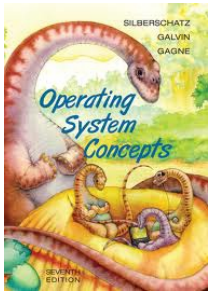
ScreenShot - 12 : About Window



ScreenShot - 13 : Problem Statement Window

Bibliography

Books



Operating System Concepts
7th edition
-Silberschatz, Galvin

Java The Complete Reference
7th Edition
- Herbert Schildt

Internet

Google India

WikipediA

The Dining Philosopher Problem