The Attitude Determination and Control Systems (ADCS) Task Scheduler

Thesis presented in fulfillment of the requirement for the degree of Master of Science in Engineering Science at the University of Stellenbosch

By M.H. Ntsimane

December 2001

Promoter: Prof. J.J du Plessis

Declaration:

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and has not previously been submitted at any university, in part or in its entirety, for any requirements towards the achievement of any degree.

M.H Ntsimane Date

Abstract

A new task scheduler for the Attitude Determination and Control System (ADCS) of the Stellenbosch University Satellite (SUNSAT) has been designed and tested on a personal computer. This new scheduler is capable of uploading new control tasks, or changing existing control tasks, on an individual basis. This is an improvement on the current ADCS task scheduler, where the control tasks are hard-coded in the scheduler, requiring the entire software image of the scheduler to be uploaded if a new task is to be added, or an existing task is to be changed.

The new scheduler was developed using the Java programming language. The Java *ClassLoader* class is used to dynamically load tasks to a linked list. The scheduler thread runs through this linked list and schedules all the tasks that have become schedulable. New tasks can be added to the list without stopping the scheduler.

The new scheduler has been successfully implemented on a personal computer, laying a good foundation for implementation in an embedded environment based on processors such as the T800 Transputer of the ADCS or the 80386 processor of the secondary onboard computer (OBC2).

Opsomming

’n Nuwe taak skeduleerder vir die oriëntasie beheerstelsel (Engels: Attitude Determination and Control System, of ADCS) van die Stellenbosch Universiteit Satelliet (SUNSAT) is ontwerp en getoets op ’n persoonlike rekenaar. Hierdie nuwe skeduleerder het die vermoeë om ekstra beheertake op te laai, of bestaande beheertake te wysig, onafhanklik van mekaar. Dit is ’n verbetering op die huidige ADCS taak skeduleerder waar take hard gekodeer is in die skeduleerder en waar vereis word dat die volledige sagteware beeld van die skeduleerder opgelaai moet word indien ’n nuwe taak bygevoeg wil word of ’n bestaande taak gewysig wil word.

Die nuwe skeduleerder is ontwikkel met behulp van die Java programmeringstaal. Die Java *ClassLoader* klas is gebruik om take dinamies te laai en te voeg by ’n skakellys. Die skeduleerder proses stap dan deur hierdie skakellys en skeduleer alle take wat skeduleerbaar geword het. Nuwe take kan by die skakellys gevoeg word sonder om die skeduleerder te stop.

Die nuwe skeduleerder is suksesvol geïmplementeer op ’n persoonlike rekenaar en lê ’n goeie grondslag vir implementering in ’n toegewyde stelsel omgewing gebaseer op byvoorbeeld die T800 Transputer van die ADCS of die 80386 verwerker van die sekondêre aanboord rekenaar (OBC2).

In the loving memory of my Grandparents and my Step Dad.

Acknowledgements

For someone who started working on this project without any background on programming, it was always going to be very difficult to get this far. I am deeply honored to have got assistance from all the people I always consulted and those who supported me all the way long. I would like to single out my Mentor Japie Engelbrecht. It was always an eye opener every time I came to you with questions. I guess I could have not made it this far if it was not for your perseverance. Additionally, I would like to thank Xandri Farr and Pierre Oosthuizen. To them I would like to say, it all started with you guys and I appreciate it. My supervisor Professor Jan du Plessis, thank you for the guidance I got every time when things were not going well.

To all my friends, thank you for the support and thanks for the good times we have had in the process. To my family, Mom Suzan, sister's Boipelo and Tsholofelo, you all inspire me. GOD BLESS YOU ALL. I LOVE YOU ALL.

Table of contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

Attitude Control processor	:	ACP
Attitude Determination and Control System	:	ADCS
Application Programming Interface	:	API
Charge Couple Device	:	CCD
Current Scheduler on SUNSAT I	:	CSS
Interface Control Processor	:	ICP
Integrated Test System	:	ITS
Java Virtual Machine	:	JVM
Linked List	:	LL
Onboard Computer	:	OBC
Onboard Computer 2	:	OBC2
Operating System	:	OS
Personal Computer	:	PC
Stellenbosch University Satellite	:	SUNSAT
Universal Asynchronous Receiver/Transmitter	:	UART
Virtual Machine	:	VM

CHAPTER 1: INTRODUCTION

The thesis concerns the implementation of an effective scheduler capable of uploading and scheduling extra control tasks and algorithms without having to upload the whole ADCS software image. A task or process is an executing program. A scheduler algorithm is responsible for determining which tasks are schedulable (i.e. tasks that have to execute) and give them access to the processor (figure 1.1). This chapter introduces the concept of scheduling. It starts by introducing the broader concept of scheduling and ultimately reaches dynamic scheduling, which is primarily what the project is concerned about.

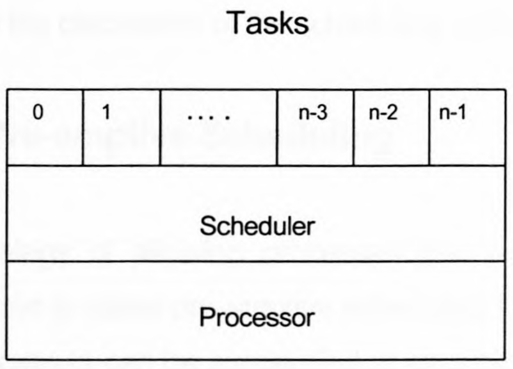


Figure 1.1: A block representation of scheduler linking tasks to the processor

1.1 Scheduling

When two or more processes are runnable, the operating system must decide which process to run first. The part of the operating system concerned with this decision is called the scheduler and it uses the scheduling algorithm. Therefore, scheduling can be defined as a process where a set of tasks is made to execute on the processor. In Java, the process involves taking a thread and executing that thread on the Java Virtual Machine (JVM). When deciding on a scheduling policy, it is very important to consider factors that make the scheduler more effective. Those include fairness to the processor, thus ensuring that the tasks are getting their fair chance on the processor. Another important thing to consider is the efficiency of the scheduler where, the processor would be kept as busy as possible. The execution time of the scheduling algorithm should be minimized as much as possible.

It is not always possible to achieve all the factors that make a good scheduler all at once because some of the factors are contradictory to each other. Also it is very important to note that tasks have different priorities, therefore some tasks will require more time to the processor hence there are different scheduling policies depending on the purpose of the scheduler. A number of scheduling policies are briefly discussed. The advantages and disadvantages of these policies are also outlined and these leads to the discussion of the scheduling policy adopted for this project.

1.1.1 Pre-emptive Scheduling

The strategy of allowing processes that are logically runnable to be temporarily suspended is called pre-emptive scheduling [1]. Pre-emptive scheduling policy means that processes can be suspended at an arbitrary instant so that other processes can be run.

An example of pre-emptive scheduling is the *Round Robin scheduling*, where tasks are each given some time interval to run and once each task's time has expired then the processor is given to another task and the task is put on the end of the list. In this scheduling policy, it is assumed that all the tasks are equally important. Another example is *priority scheduling*, where tasks are given priorities depending on their importance and then tasks with the highest priority are allowed to run first. To prevent high-priority tasks from running indefinitely, the scheduler may decrease the priority of the running tasks at intervals. If the priority of that task drops to below that of the next highest process, then task switching can occur. Alternatively, each task may be allowed a maximum time interval that it is allowed to hold on to the processor continuously. When this time interval is used up, the next highest priority task is given a chance to run. Another interesting approach would be *Guaranteed Scheduling* where if there are *n* tasks then *1/n* time of the processor is given to the task. Some tasks may never be completed within their allocated time and that means there might be no output for some tasks.

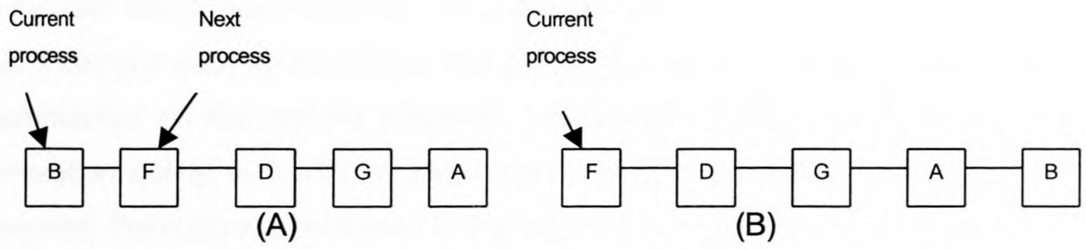


Figure 1.2: An example of Round robin scheduling (Pre-emptive scheduling).
(A) A list of runnable processes.
(B) The list of runnable processes after B uses up its quantum [1]

1.1.2 Non pre-emptive Scheduling

This type of scheduling policy allows a task that has the processor to run as long as it wants or until it has completed. This is very effective policy if the user already knows what type of tasks are going to be scheduled and how long will the tasks take to complete.

An example of non pre-emptive scheduling policy is the *Run-to-completion scheduling*. This type of scheduling, as the name suggest allows the task to hold on the processor until it has finished, once it is running. This policy is simpler to implement as once the task is being executed, it holds on to the processor until it has finished executing. However since a task holds on to the processor until it has finished running, if a task can go into an infinite loop, other tasks will never access the processor. Also if there are time critical tasks, then this policy is not a good choice as time critical tasks will miss their deadlines whilst other tasks are holding on to the processor.

1.1.3 Why Run-to-completion scheduling?

Run-to-completion scheduling allows a task that is running to hold on to the processor until it is through. In the run-to-completion policy, there is no need for condition checks as the task simply gets complete use of the processor and returns it once it is through, thus making it easy to implement this policy. The run-to-completion policy has been implemented on the current scheduler on Sunsat I (CSS). An example of run-to-completion policy is having a loop executing at a pre-defined rate. As the loop executes, the methods contained in the loop are executed one after the other. When one method is being executed, the other method is never executed until the previous method has finished executing.

The project requires implementation of an effective scheduler capable of uploading and scheduling extra tasks without having to upload the whole Attitude Determination and Control System (ADCS) software. Since the current scheduler is capable of effectively scheduling tasks, the scheduling policy used is adopted for this project. Also because ADCS tasks are simple algorithms, this makes it easy to implement the run-to-completion policy. The ADCS tasks are time-critical but there are enough resources to see to it that tasks are completed in time. By implementing a watchdog thread ensures that tasks cannot hold on to the processor forever at the same time making sure that not many tasks miss their deadlines.

1.2 Dynamic Scheduling

Section 1.1 introduced scheduling and described a few scheduling policies that could be used during scheduler implementation. Dynamic scheduling achieves scheduling tasks without first knowing about them. Unlike in the current scheduler where tasks are embedded in the main loop, dynamic scheduling involves invoking tasks when they are supposed to be scheduled. The drawback in the current scheduler comes when there are new tasks that should be added to the scheduler, then the whole scheduler software should be re-compiled and re-loaded. The aim of this new scheduler is to make provision for tasks that will be created at a later stage, while the scheduler is already running. The new scheduler should be able to dynamically load tasks into the scheduler and dynamically link them to the scheduler application.

The advantages of dynamic scheduling are,

- Only tasks created when the scheduler is already running are loaded separately. There is no need to upload the whole software image when adding new tasks to the scheduler.
- Tasks can be added to the scheduler at anytime without first stopping the scheduler.

1.3 Thesis overview

The project concerns implementation of an effective scheduler capable of uploading and scheduling extra control tasks and algorithms without having to uploading the whole ADCS software image. The Onboard Computer 2 (OBC2) based attitude control computer, is responsible for all the closed-loop control algorithms on the ITS. The approach to the project involved uploading techniques where class loading was used. Class loading helped us to be able to load different tasks in a linked list (LL) where they are scheduled. Run-to-completion scheduling policy was used and this is explained in section 1.1.3. Throughout this thesis report, the term new scheduler means the scheduler that is developed in this project and the current scheduler implies the scheduler currently on SUNSAT I.

In summary, tasks are loaded using class loader and placed in a LL. Then the scheduler will run through the LL and schedule the tasks depending on whether their respective schedulable flags are set.

- Chapter 2 introduces the theory background used in this thesis. The concepts covered are class loading (for uploading tasks), the ADCS tasks to be scheduled and then introduces the current scheduler before suggesting the design of the new scheduler.
- Chapter 3 discusses the software development of the new scheduler. The focus is the design of each object used for the scheduler design and then the implementation software.
- Chapter 4 describes the evaluation and testing process of the software and determines if the objectives were met. The methods used for evaluation and testing are outlined in this chapter.
- Chapter 5 is the conclusion, where the contribution made by this thesis is evaluated and a few recommendations are proposed for possible future research. Then the shortcomings of the new scheduler are briefly touched on.

1.4 Summary

This chapter introduced the concept of scheduling and highlighted the need for a new scheduler. The new scheduler comes as a result of drawback on the CSS, therefore the current scheduler is also looked at. The next chapter takes a look at the theoretical background on the concepts adopted in this project.

CHAPTER 2: BACKGROUND

The previous chapter started by introducing the concept of scheduling. This chapter investigates the supporting theories in a run-up to the design of the new scheduler.

2.1 Class loading

Class loading involves loading class files into the virtual machine, where the bytecodes they contain are executed by the execution engine [4]. After writing a Java program, it is compiled to a class file and it is the class-file that is loaded to the JVM and executed in the execution engine. An application can need extra classes that implement Java Application Programming Interface (API). Only class files from the Java API needed by the running program will be loaded into the virtual machine. Figure 2.1 shows how class files and the Java API's that are required are loaded and executed. Java applications can use two types of class loaders: a "Primordial" class loader and "user-defined" class loader discussed in sections 2.4.1 and 2.4.2 respectively.

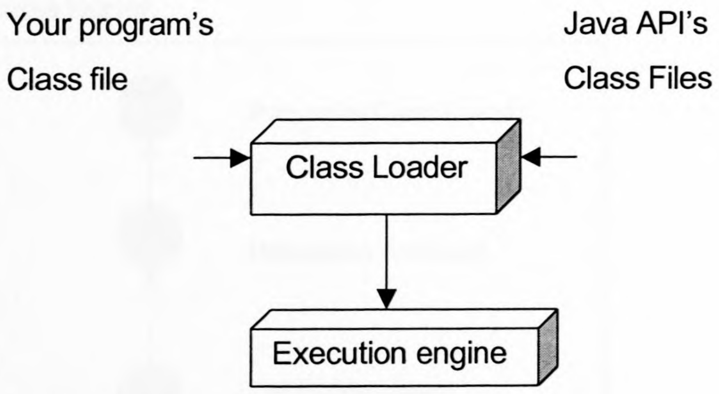


Figure 2.1: A basic block diagram of the JVM

Main roles of class loading are summarised below [5]:

- Loads class files into the Virtual Machine
- Identifies the package to which a loaded class belongs
- Locates and loads any classes referenced by the currently loaded class

- Verifies attempts by the loaded class to access classes outside its package
- Keep track of the sources of loaded classes, and makes sure that the classes are loaded from valid sources

2.1.1 Parent-Child Delegation Chain model

The delegation model for loading classes is described as below. The basic idea is that every class loader has a "parent" class loader. When one class loader is asked to load a class, it either loads the class itself or asks another class loader to do so. In other words, the first class loader can delegate to the second class loader. Since every class has to have a parent, depending on the constructor used, you can either specify the parent or use a default parent. To specify the parent use the constructor, *protected ClassLoader (ClassLoader parent)* and to use the default constructor use *protected ClassLoader ()*. The Primordial class loader is the root ancestor of all class loaders [6]. Figure 2.2 shows a delegation process of a ClassLoader requesting its parent to load a class, who in turn passes the request to its parent until the primordial class loader level. NOTE: Any class loader can load the requested class, so the chain can be broken at any level between the primordial class loader and the requesting class loader.

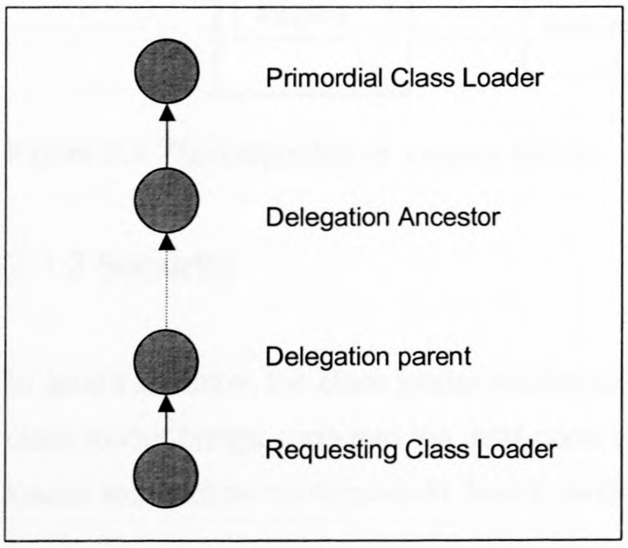


Figure 2.2: Class Loader searching for classes

2.1.2 Type Loading, linking and initialization

The JVM makes types available to the running program through a process of loading, linking and initialization [7]. Loading is the process of bringing a binary form of a type into the JVM, while linking is the process of incorporating the binary type data into the run-time state of the Virtual Machine (VM). Linking is divided into three sub-steps: verification, preparation and resolution. Verification ensures that the type is properly formed and is fit for use by the JVM. Preparation is the allocation of memory needed by the type, such as memory for any class variables. Resolution is the process of transforming symbolic references in the constant pool into direct references. This sub-step can be delayed until each symbolic reference is actually used by the running program. Then during initialization, class variables are given their proper initial values.

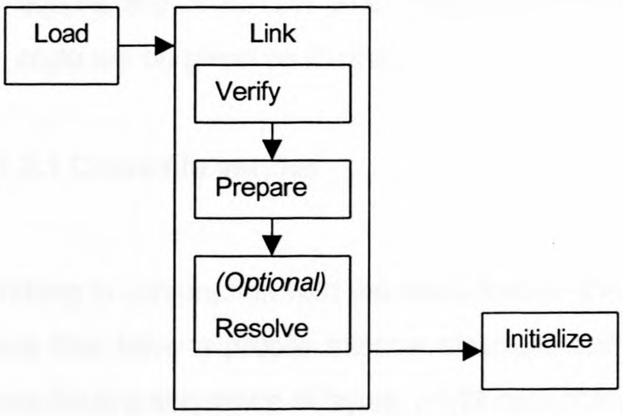


Figure 2.3: The beginning of a type's lifetime

2.1.3 Security

In Java's sandbox, the class loader architecture is the first line of defense. After all, the class loader brings code into the JVM-code that could be hostile or buggy. The class loader architecture contributes to Java's sandbox in three ways [8]

- Preventing malicious code from interfering with the benevolent code by providing name spaces (*name space is a set of unique names-one name for each loaded* *class- that the JVM maintains for each class loader*) for classes loaded by different class loaders. Name spaces contribute to security, because you can place a shield between classes loaded into different name spaces. Classes in the same name space can interact with each other but classes in different name spaces cannot even detect each other's presence unless a mechanism that enables them to interact is explicitly provided.

- Guarding the borders of the trusted class libraries by making it possible for trusted packages to be loaded with different class loaders compared to untrusted packages. You can grant special access privileges between types belonging to the same package by giving members protected or package access. This special access is granted to members of the same package at runtime-only if the same class loader loaded them.
- Placing code into categories (*called protection domains*) that will determine which actions the code can take. Protection domains will define what permissions the code will be given as it runs.

2.1.3.1 Class-file verifier

Working in conjunction with the class loader, the class verifier ensures that the loaded class files have a proper internal structure and that they are consistent. Because a class-file is a sequence of bytes, a VM cannot know whether a particular class-file was generated by a well-meaning Java compiler or by shady crackers who were bent on compromising the integrity of the VM. As a consequence, all JVM implementation have a class-file verifier that can be invoked on class files to make sure that the types they define are safe to use. Class-file verifier of the JVM does most checking before bytecodes are executed.

The class-file verifier operates in four distinct passes. During pass one, which takes place as a file is loaded the class-file verifier checks the internal structure of the class-file to make sure that it is safe to parse. During pass two and three, which takes place during linking, the class-file verifier makes sure that the type data obeys the semantics of the Java programming language, including verifying the integrity of any bytecodes it contains. During pass four, which takes place, as symbolic references are resolved in the process of dynamic linking, the class-file verifier confirms the existence of symbolically referenced classes, fields and methods. Also during pass four, the class-file verifier look at classes that refer to one another to make sure that they are compatible since Java programs are dynamically linked. Java compilers will often recompile classes that depend on a class you have changed, and in doing so, they will detect any incompatibility at compile time.

2.1.3.2 Safety features on the JVM

Once the JVM has loaded a class and has performed passes one through three of class-file verification, the byte codes are ready to be executed. Besides the verification of symbolic references, JVM has several other built-in security mechanisms operating as bytecodes are executed [9]. Here are some of the security features implemented on the JVM.

- JVM makes the Java programs more robust and makes their execution more secure by granting a Java program-only type safe, which provides structured ways to access memory.
- To back up the above security feature, is the unspecified manner in which the run-time data areas are laid out inside the JVM. When loading a class, the JVM decides where in its internal memory to put the bytecodes and other data it parses from the class file. When starting a thread, it decides where to put the Java stack it creates from the stack. And when creating an object, it decides where in memory to put the object. This makes it difficult to predict by looking at a class-file where in memory the data representing that class, or objects instantiated from that class, will be kept.
- The structured error handling with exceptions is another mechanism built into JVM that contributes to security. It is because of this, when a security violation occurs, instead of the system crashing, exception is thrown or an error is thrown. Throwing an exception could result in the death of the thread but is often used as a way to transfer control from the point in the program where an exception condition arose to the point in the program where the exception is handled.

2.1.4 Summary to class-loading API

Since this project introduced the concept of class loading, some steps followed when class loading happens are discussed. Below, are some important factors that happen during class loading [10]

- Calls *findLoadedClass ()* to check if the class has already been loaded.
- If the current class loader has a specified delegation parent, call the *loadClass ()* method of the parent to load the class. Otherwise, call the *findSystemClass ()* method to see whether the class can be found among system classes.
- Call the *findClass ()* method to find the class.

Below we look at different class loaders and their contribution to the class loader chain.

2.1.4.1 Primordial class loader

Since each class is loaded by its defining class loader and also a class loader is itself a class which must be loaded by another class loader, a chicken-and-egg question arises, that is from where does the first class loader come. The answer to this is a primordial class loader that bootstraps the class loading process [11].

Primordial class loader is part of the JVM implementation. I.e. if a JVM is implemented as a C program on top of an existing Operating System (OS), then the Primordial class loader will be part of the C program. Primordial class loader is also known as a primordial class loader, system class loader, or default class loader. A Primordial class loader is responsible for loading in the Java runtime. It is the "root" in the class loader hierarchy. The system class loader is a descendant of the Primordial class loader. It is responsible for loading in the application, as well as for loading classes and resources in the application's classpath.

2.1.4.2 User-defined class loader

User-defined class loaders are written in Java, compiled to class files, loaded into the JVM, and instantiated just like any other object. The conclusion is therefore that they are just another part of the executable code of a running Java application. One advantage of user-defined class loaders is that, at compile time you do not have to know the all the classes that may ultimately take part in a running Java application i.e. user-defined class loaders allow us to dynamically extend a Java application at run time. Through user-defined class loaders, an application can load and dynamically link to classes and interfaces that were unknown or did not even exist during application compilation [12].

Often user-defined class loaders relies on other class loaders, at the least, upon the class loaders created at virtual machine start-up, to help it fulfill some of the class load request that come to its way. Therefore this is where the parent-child delegation model is useful.

2.1.4.3 Dynamic Linking and extension

When Java programs are compiled, a separate class file for each class or interface in the program are created. Although the individual class files might appear independent, they actually harbor symbolic connections to one another and to the class files of the Java API. As a program runs, the JVM builds an internal web of interconnected classes and interfaces and hooks them together in the process of dynamic linking [13]. Java's architecture enables its programs to be dynamically extended which refers to the run-time process of deciding other types to use, loading them, and using them. In addition to simply linking types at run time, Java applications can decide at run time, which types to link.

The method used to dynamically extend a Java program/application is either *forName ()* method from class *java.lang.class* or *loadClass ()* method of an instance of user defined class loader, which can be created from any subclass of *java.lang.Classloader*. The *forName ()* method is used if there are no special needs and is straightforward approach to dynamic extension. However, the *loadClass ()* method can help meet the needs that the *forName ()* cannot meet (e.g. Loading classes such as by downloading them across a network, retrieving them from a database, extracting them from encrypted files or even generating them on the fly). For security needs, where loaded types are placed into protected domains the *forName()* method is useful.

2.2 Attitude Determination and Control System (ADCS)

The attitude determination and control system (ADCS) stabilizes the vehicle and orients it in desired directions during the mission despite external disturbance torques acting on it [14]. Depending on the mission, it could be very important to have a highly accurate attitude determination and control because a satellite is designed to perform some specific tasks at predefined locations. An instance of this is when a satellite has to take images of a particular place at a specific time. Besides these types of tasks, the satellite can also be used for communications and other purposes, and this requires that the satellite be oriented in a certain positions for those tasks.

ADCS is defined in terms of attitude determination, where the satellite orientation is determined and attitude control where the satellite controlled to a specified, predefined orientation. For attitude determination, the satellite uses the sensors and for orientation it uses the actuators. The aim of the ADCS task scheduler is to schedule attitude determination algorithms that use information from these sensors to determine the orientation of the satellite. After determining the orientation of the satellite, then the ADCS task scheduler will schedule control algorithms to orient the satellite in a specified, predetermined direction.

This section investigates the current ADCS tasks implemented for the attitude determination and control of SUNSAT.

2.2.1 Actuators

Attitude control is the process of achieving and maintaining and orientation in space. Control torques, are generated intentionally to control the attitude of the satellite. The control hardware or actuator is the mechanism that supplies the control torque [15]. Modification of the satellite orientation is generally obtained by generating a torque which, taking account of the dynamics of the particular satellite, causes an angular acceleration, or velocity, about an axis [16]. In the current SUNSAT system, attitude sensors send attitude data to an onboard computer, which determines the attitude and then activates the actuators to control the satellite to the required orientation. Below are the control systems of the current SUNSAT system.

2.2.1.1 Reaction wheels

Reaction wheels are essentially torque motors with high-inertia rotors. They can spin in either direction, and provide one axis of control for each wheel. Momentum wheels are reaction wheels with a nominal spin rate above zero to provide a nearly constant angular momentum.

In SUNSAT, there are four reaction wheels, one aligned to each of the X, Y and Z body axes and a fourth also aligned to the Z-axis to add redundancy. Conventional DC motors drive two of the wheels and the other two are driven by the brushless DC motors. The main control function of these wheels is to provide accurate pointing and tracking control during imaging and also for performing large angular slew maneuvers to point the imager at different targets within a short span of time [17].

2.2.1.2 Magnetic Torquers

In spacecraft, magnetic torquers are used as actuation devices. These torquers use magnetic coils or electromagnets to generate magnetic dipole moments. Magnetic torquers can compensate for the spacecraft's residual magnetic fields or attitude drift from minor disturbance torques. They can also desaturate momentum-exchange systems but usually require much more time than thrusters. A magnetic torquer produces torque proportional (and perpendicular) to the earth's varying magnetic field [18].

In SUNSAT, the magnetic torquers are six air core coils wound into recesses around the solar panels and in the top plate. Using these coils, a magnetic dipole moment is generated in three axes. The interaction of this magnetic dipole moment with the geomagnetic field result in magnetic torque, which is applied to the satellite body. Therefore with appropriate control algorithms, this magnetic torque may be utilized for the attitude control [19].

2.2.1.3 Gravity Boom

The deployable gravity boom provides passive attitude control by utilising the gravity gradient torque, which acts upon the satellite body. After SUNSAT was released into its orbit by the launch vehicle, it was detumbled and controlled to track a pre-calculated pitch rate using the magnetic torquers. The boom was then deployed at the correct moment to result in a nadir pointing gravity gradient lock [19].

2.2.2 Sensors

The goal of attitude determination is to determine the orientation of the spacecraft relative to either an inertial reference frame or some specific object of interest, such as the earth [20]. In the current SUNSAT system, the sensors are used to determine the orientation of the satellite, and include six solar cells, a three-axis flux-gate magnetometer, a sun angle sensor, two horizon sensors and a star sensor [21]. As the satellite is on orbit, to determine its orientation it uses the sensors and uses known information to determine the correct orientation.

The aim of the ADCS task scheduler is to schedule algorithms that use information from these sensors to determine the orientation of the satellite. After determining the orientation of the satellite, then the ADCS task scheduler will schedule control algorithms to orient the satellite in a specified, predetermined direction.

There are a number of sensors implemented on SUNSAT that are used in this regard. This section gives a brief description of sensors implemented on SUNSAT.

2.2.2.1 Sun Angle Sensors

Sun sensors are visible-light detectors, which measure one or two angles between their mounting base and incident sunlight. They are popular, accurate and reliable, but require clear field of view. They can be used as part of normal attitude determination system, part of the initial acquisition or of failure recovery system, or part of an independent solar array orientation system [22].

The current SUNSAT system uses Sun Angle Sensors that measures the azimuth angle within a ±60° field of view to an accuracy of 1mrad. The sensor uses similar Charge Couple Device (CCD) technology to the horizon sensors and the sensor head consists of a slit aperture perpendicular to the linear CCD. Yaw attitude angles may be obtained from the sun sensor, but its measurements are only valid for a valid field of view and a valid horizon illumination [23].

2.2.2.2 Horizon Sensors

They are infrared devices that detect the contrast between the cold of the deep space and the heat of the earth's atmosphere. They provide Earth-relative information directly for Earth-pointing spacecraft, which may simplify onboard processing [24]. SUNSAT scheduler schedules algorithms that obtain measurements from these sensors to determine its orientation.

SUNSAT 's horizon sensors are linear CCD and lens assemblies, which look below the local horizon level with a ±15° field of view. Pitch and roll attitude angles to an accuracy of 0.5 mrad may be measured with these sensors [23].

2.2.2.3 Star Sensors

An image of a given portion of the sky provides a map of stars whose relative positions are detected and compared with a reference map [25].

They use a pixel matrix CCD to take an image of the star field within a 10°×10° field of view. Individual stars are then extracted from this image and constellations are identified and using a catalogue, based on the magnitudes and separation distances of the stars detected in the image. This information can be used to determine the roll and yaw of the satellite to the accuracy of 0.5 mrad and pitch angles may be estimated to an accuracy, which depends on the star separation distance [23].

2.2.2.4 Sun Cell Sensors

They are six cosine-law solar cells mounted on each facet of SUNSAT's cubic body. Using a satellite orbit propagator and a sun model, full attitude information can be obtained from these sensors to an accuracy of ±5°. The sun vector with respect to the satellite body may be obtained from the short circuit currents of the solar cells. The temperatures of the cells are also measured to make sensitivity corrections to the current-measurements [26].

2.2.2.5 Magnetometer

The magnetometer measures the strength and direction of the geomagnetic field vector in three axes. The measured geomagnetic field may be compared to geomagnetic field models to obtain attitude information and may also be used to estimate the torquer produced by the magnetic torquers' [23].

2.3 Overview of the current SUNSAT scheduler

The current Stellenbosch University Satellite (SUNSAT) scheduler executes as a loop in the Attitude Control Processor (ACP) software. The main execution loop communicates to the Interface Control processor (ICP) communication thread every second. If it takes longer than one second, then the watchdog timer thread will set the time-out flag. Then the Universal Asynchronous Receiver/Transmitter (UART) will be reinitialized. ICP communications will reset the current and previous magnetometer measurements to zero, disable the magnetic torquer control algorithm and clear reaction wheel on flag. Unless of course specified, all the tasks performed in the main loop are one-second tasks. The flow chart as described in [2] is attached at appendix A. In this sub-section, focus is on the current SUNSAT scheduler and its execution.

2.3.1 Overview of the ACP software implementation

The current SUNSAT scheduler is included in the ACP software implementation. In order to discuss the CSS it is then important to discuss the ACP software implementation. The execution of the ACP software as outlined in [3] consists of the following processes:

- The main execution loop,
- A watchdog timer thread,
- A thread to receive Onboard Computer (OBC) communications packets on the links,
- And a thread to receive ICP communications on the UART.

These four processes are synchronized by way of a semaphore. The main execution loop executes until it reaches the semaphore and then waits for one of the three threads to signal the semaphore. When the ACP is running in its normal real-time mode, the ICP will send a data packet to the ACP once every second, causing the ICP thread to signal the semaphore. If in simulation mode, the OBC will send simulated sensor packets at a much higher rate than once per second, causing the

OBC thread to signal semaphore. If both ICP thread and the OBC thread fail to signal the semaphore, the watchdog timer thread will time out after 1.2 seconds and signal the semaphore, allowing the main loop to resume execution.

It is therefore important to have a look at these threads to understand their importance in the implementation of ACP software. The next subsection looks at these threads and gives a brief description on them.

2.3.2 The Watchdog timer thread

If the watchdog timer is not reset by the main execution loop, it times out after 1.2 seconds. When it is timed out, it increments the time since epoch, sets the time out flag and signals the semaphore in the main loop. The semaphore is signaled to allow the main execution loop to resume if either the OBC thread (in simulation mode) or the ICP thread (in real-time mode) has failed to signal the semaphore.

2.3.3 The OBC Thread

The OBC thread is responsible for receiving OBC communication packets from the links. If a valid OBC packet has been received, a packet received flag will be set to indicate to the main loop that a packet is available to have its data extracted. Upon reception of any packet, an acknowledgement/not acknowledge variable is assigned a value of "acknowledge" if the received packet had a valid checksum, or a value of "not acknowledge" if the packet had an invalid checksum. This variable tells the main loop to respond to the received packet by sending either acknowledge or a not acknowledge message to the OBC. If the ACP is running in simulation mode without the ICP, the OBC thread will also signal the semaphore in the main loop when a valid simulated sensor packet has been received.

2.3.4 The ICP thread

The ICP thread is responsible for receiving the ICP communication packet from the

UART. If the ICP packet has a valid end byte, the ICP thread extracts the data from the packet and signals the semaphore in the main loop.

2.3.5 Block representation of the current SUNSAT scheduler

SUNSAT has ACP on OBC2, which handles its attitude determination and control. The OBC2 is linked to the attitude determination and control hardware (i.e. Sensors and actuators). Data from the sensors is extracted in the form of OBC packets and is used by either the ICP thread or the OBC thread for attitude determination of the satellite. After attitude determination, data is transmitted to the actuators to orient the satellite to the desired position. Two figures below (Figure 2.4 and figure 2.5) gives block representations on how the actuators and sensors are linked to OBC2 and ACP.

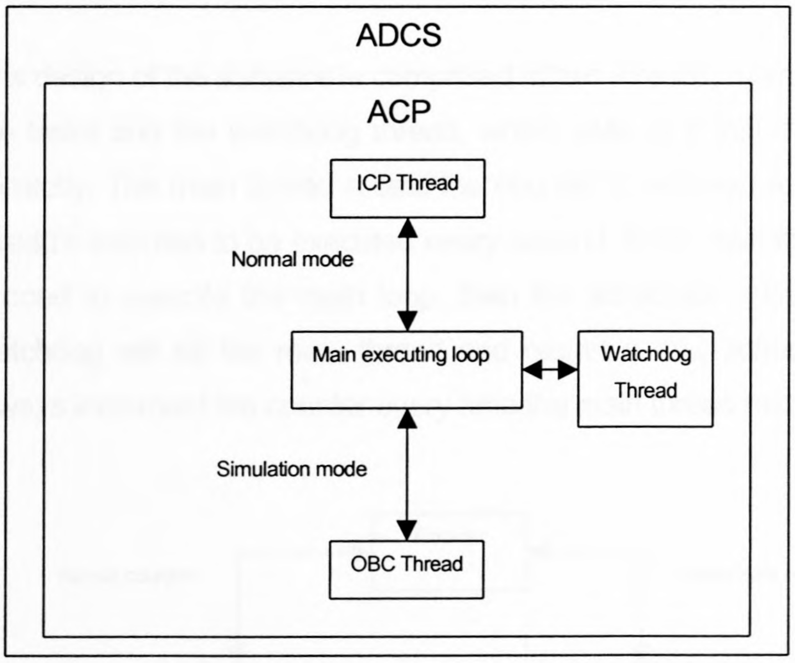


Figure 2.4: A block representation of main loop executing the threads

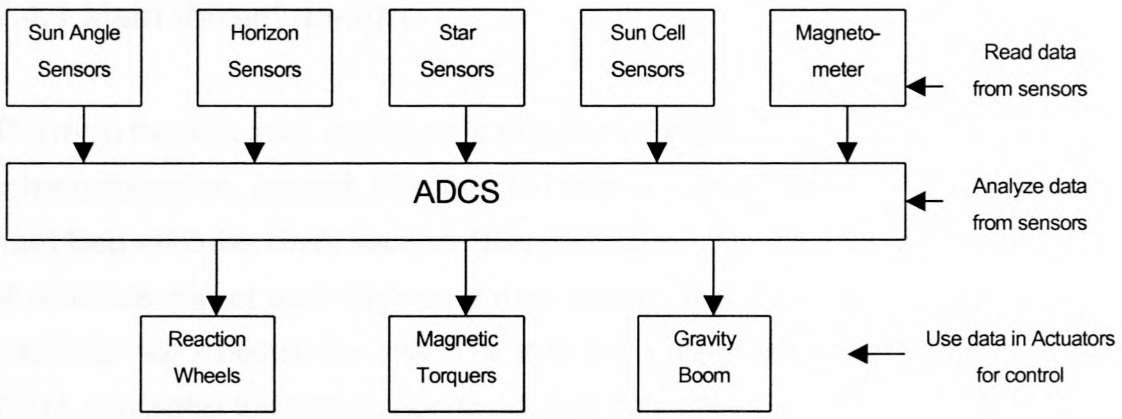


Figure 2.5: A block representation of ADCS linked to ADCS computer

2.4 Design

The design of the software is comprised of two threads, main thread, which schedules the tasks and the watchdog thread, which sees to it that the main thread executes correctly. The main thread resets the counter in between executing tasks. The main thread‘s loop has to be executed every second. If the main thread takes longer than a second to execute the main loop, then the scheduler risks skipping tasks, and the watchdog will kill the main thread and restart a new scheduler. The watchdog will always increment the counter every time the main thread restarts the counter.

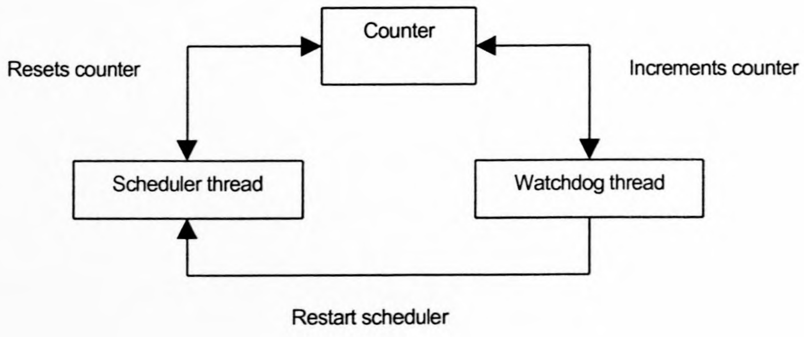


Figure 2.6: Block description of the new scheduler

2.4.1 Main thread design

The main thread's main execution is to schedule the tasks. The most frequent task is schedulable every second, therefore to make sure that each task is scheduled, the main loop executes every second. During execution, the main thread will always look at schedule flag of each task as it runs through the LL. If the flag is set then the scheduler will schedule the task. The *load tasks* object is used to load new tasks to the LL containing the schedulable tasks. The data object is passed as a parameter to many tasks, hence it is so frequently scheduled.

To ensure that the scheduler does not give one task the processor forever, the counter (Counter is an integer value in the counter object) will always be reset by the main thread every time the thread runs. If one task is executing for a long time (either the task is an infinite loop or a task takes a long time on the processor), then the watchdog will ensure that the main thread is killed and a new thread is created. The watchdog thread is discussed in the next sub-section. Figure 2.3 gives a flow execution of the main thread.

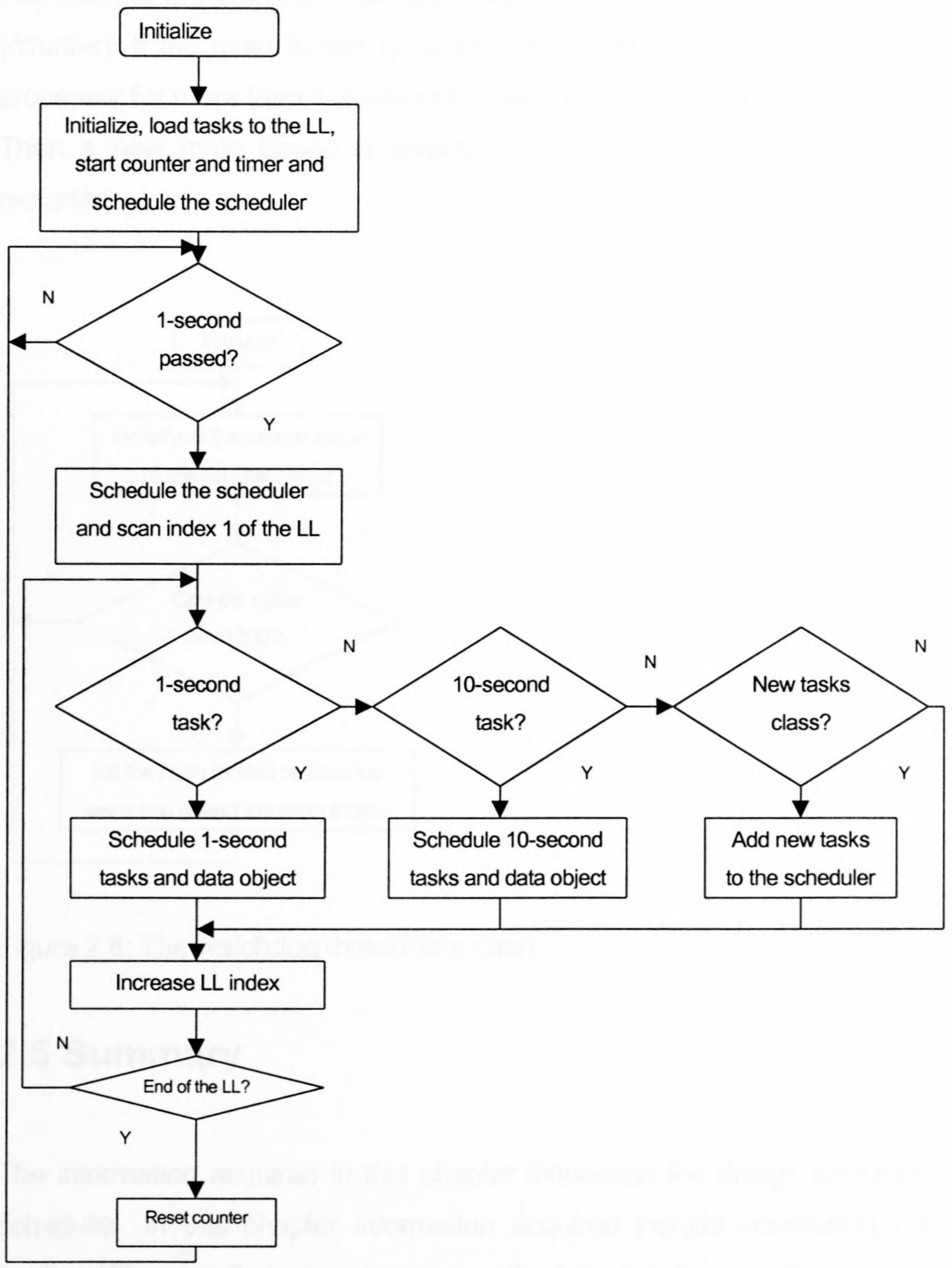

Initialize
Initialize, load tasks to the LL, start counter and timer and schedule the scheduler
1-second passed?
N
Y
Schedule the scheduler and scan index 1 of the LL
1-second task?
10-second task?
New tasks class?
N
N
N
Y
Y
Y
Schedule 1-second tasks and data object
Schedule 10-second tasks and data object
Add new tasks to the scheduler
Increase LL index
End of the LL?
N
Y
Reset counter

Figure 2.7: The main-loop execution flow chart

2.4.2 Watch dog thread design

The watchdog thread ensures that the main thread executes as efficient as possible.

The counter is incremented as the thread runs and the main thread always resets it (counter). If the main thread goes into an infinite loop or one task holds on to the processor for more than 1.2 seconds, then the watchdog thread kills the main thread. Then a new main thread is created and tasks are re-loaded and the thread is restarted.

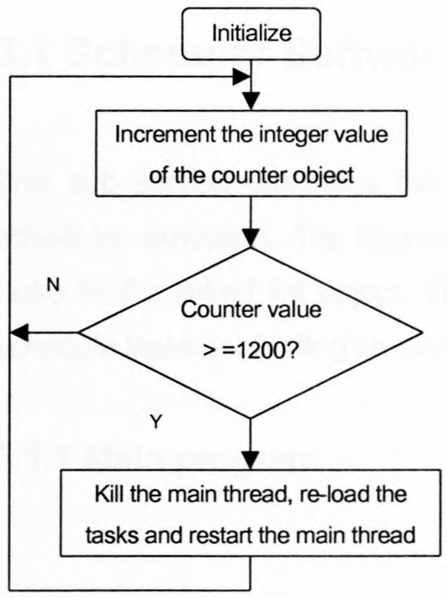

Initialize
Increment the integer value of the counter object
N
Counter value > =1200?
Y
Kill the main thread, re-load the tasks and restart the main thread

Figure 2.8: The watchdog thread flow chart

2.5 Summary

The information acquired in this chapter influenced the design structure of the new scheduler. In this chapter information acquired include scheduling policies, class loading (Class loading which involves gathering a lot of information on the Java Virtual Machine (JVM) implementation) and a bit of description of tasks that were scheduled by the current scheduler. As discussed in section 2.2.3 where the choice of the scheduling policy is discussed, the next chapter investigates the design of the new scheduler in details.

CHAPTER 3: SOFTWARE DEVELOPMENT

The chapter covers the design of the new scheduler and an investigation in the implementation of the scheduler software. Since the project has been done using Java programming language, the objects designed are also investigated in this chapter.

3.1 Scheduler Software Design

This sub section describes the scheduler software and the objects used for the scheduler execution. The figures 3.1 and 3.2 show the class loader object loading tasks to the linked list object. The scheduler then runs through the linked list and schedule tasks depending on whether they are due for scheduling.

3.1.1 Main program

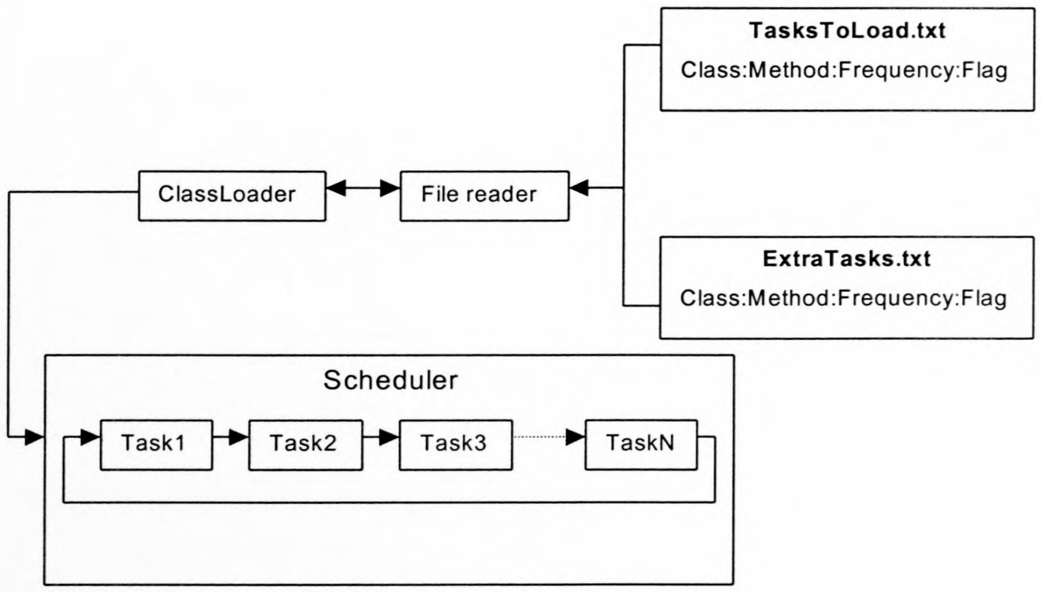


Figure 3.1: Block execution for the main program

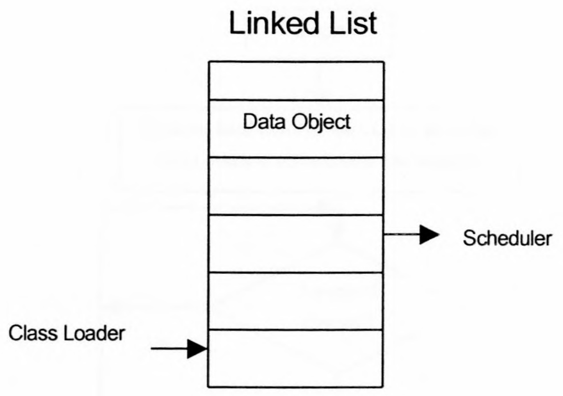


Figure 3.2: Tasks loaded to the LL and executed by the Scheduler

The block diagram below shows the execution of the scheduler when it starts to execute until tasks are executed. The class loader will load tasks read from text file *"TasksToLoad.txt"* by the file reader. Then the tasks will be placed in a LL where they will be scheduled by the scheduler. New tasks will be loaded from text file *"ExtraTasks.txt"* and the same process as above will be followed. All the new tasks will be added at the end of the LL. Figure 3.3 is the flow diagram for the main program.

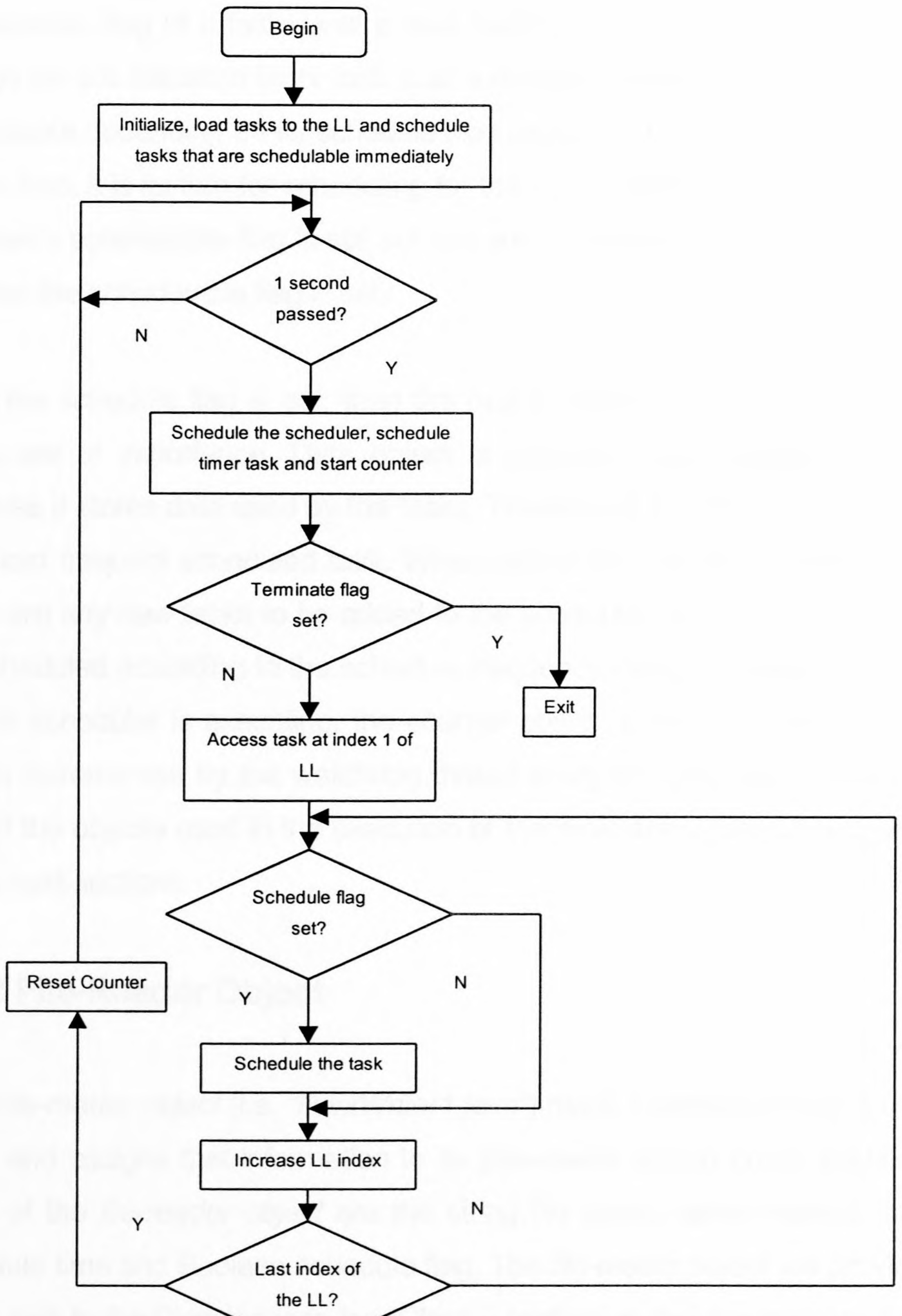

Begin
Initialize, load tasks to the LL and schedule tasks that are schedulable immediately
1 second passed?
N
Y
Schedule the scheduler, schedule timer task and start counter
Terminate flag set?
Y
N
Exit
Access task at index 1 of LL
Schedule flag set?
N
Y
Reset Counter
Schedule the task
Increase LL index
Y
N
Last index of the LL?

Figure 3.3: The main program flow diagram

The main program executes as the main thread and is executed after every second. The main loop is executing continuously and the timer task is scheduled every second where the timer task runs through the LL and determines a task's schedule frequency and sets its schedulable flag if the task is due for scheduling. The timer task object (Section 3.1.5 is the detailed description of the timer task class object) sets the schedulable flag of a task. Until a task has finished executing, its schedule flag will always be set. Because timer task is an independent object, it will always set the task executable depending on its schedule frequency. If a task is still executing and at the same time, it is its turn for scheduling for the next duration, the timer task will find that the task's schedulable flag is still set and will therefore an error is logged in the error file else the schedulable flag is set.

After the schedule flag is set, then the task is ready to be scheduled. A number of tasks are of importance. Data object is passed as a parameter by many tasks because it stores data used by the tasks. Therefore it is scheduled every second with the most frequent scheduled task. When scheduled, the new tasks object checks if there are any new tasks to be added to the scheduler. Then the rest of the tasks will be scheduled according to the schedule frequency indicated when the task is loaded. As the scheduler is executing, the counter object 's counter (integer value) is reset and is incremented by the watchdog thread every time the watchdog executes. The rest of the objects used in the execution of the main thread are discussed individually in the next sections.

3.1.2 File-Reader Object

The *File-reader object* (i.e. *"PrintWriter1.java"*) reads information from a text file about tasks and assigns that information to its *(file-reader object)* public fields. The public fields of the *file-reader object* are the string file name, string method name, integer schedule time and Boolean schedule flag. The *file-reader object* will provide the name of the task to the Classloader's *loadClass ()* method so that the task can be loaded.

File-Reader object description

The *file-reader object* has three methods *openFile (string fileName)*, *readingFile (string fileName)* and *closeFile (string fileName)*. The *openFile (string fileName)* method uses the *fileReader (string fileName)* constructor to create a new file reader, given the name of the file to read from.

The *readingFile (string filename)* method uses the *readLine ()* method to identify the line to read from the text file. Then the string tokenizer class is used to break the strings from the text file into tokens. The *nextToken ()* method of the string tokenizer is used to return the next token from this string Tokenizer. This string tokens are then casted to string for filename, string for method name, integer for schedule frequency and Boolean for schedulable flag respectively. They are then used to give information about the loaded tasks. Once the file reader reaches the end of the text file, the *close ()* method is called and closes the text file to indicate that there is no more new text in the file.

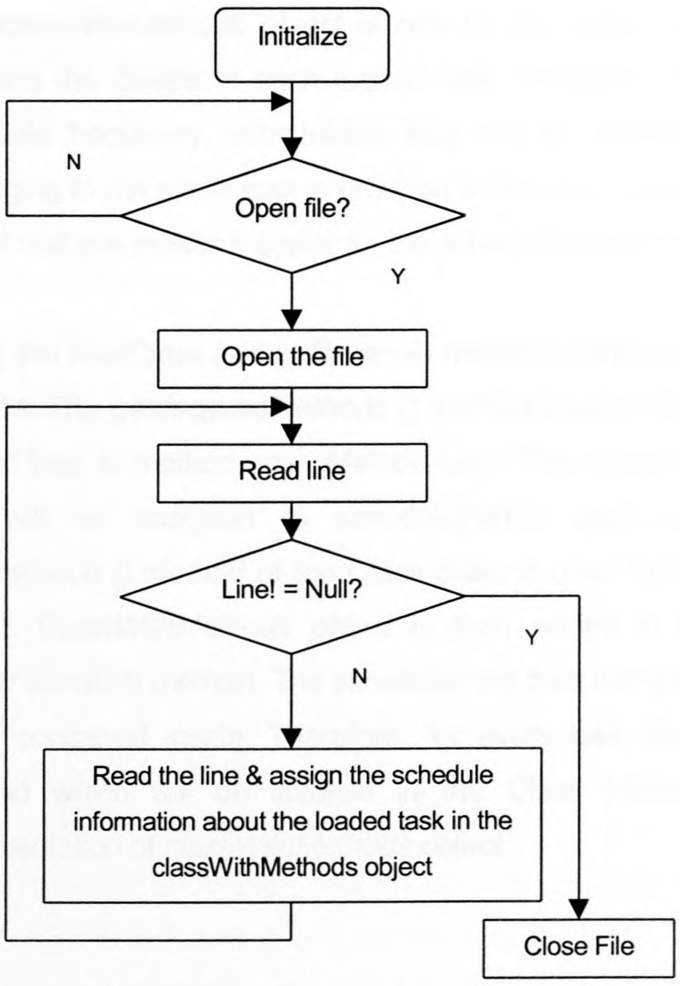

Initialize
N
Open file?
Y
Open the file
Read line
Line! = Null?
Y
N
Read the line & assign the schedule information about the loaded task in the classWithMethods object
Close File

Figure 3.4: The file reader flowchart

3.1.3 ClassWithMethods Object

ClassWithMethods object (i.e. *"classWithMethods.java"*) contains the schedule information about the tasks. Every task that will be scheduled will have its schedule information loaded in this object. When loading a task for scheduling, the task will have a name, schedule frequency and it's schedulable flag. It is ultimately in *classWithMethods object* where this information is loaded and an instance of this object is called and scheduled according to the schedule information it contains.

ClassWithMethods object description

The *classWithMethods object* is one of the fundamentals in this application as it contains the details of each loaded task. It contains the name of the loaded task, schedule frequency, schedulable flag and an instance to this object. Information belonging to the same task is grouped in the same instance of the *classWithMethods object* and this makes it easier for the scheduler to access information.

Using the *loadClass (string filename)* method of the ClassLoader to load the class to the VM. The *getdeclaredMethods ()* method assigns the schedulable methods of the loaded task to method array MethodArray. The schedule frequency and schedulable flag will be assigned to schedulePeriod and schedulable respectively. The *newInstance ()* method of the Class class is used to create the new instance of the object. *ClassWithMethods object* is then added to the LL using *add (int index, Object element)* method. The scheduler will then run through this list when scheduling tasks contained inside. Therefore, for every task loaded, a new instance will be created which will be invoked in the Class Method. Figure 3.5 is the block representation of *classWithMethods object*.

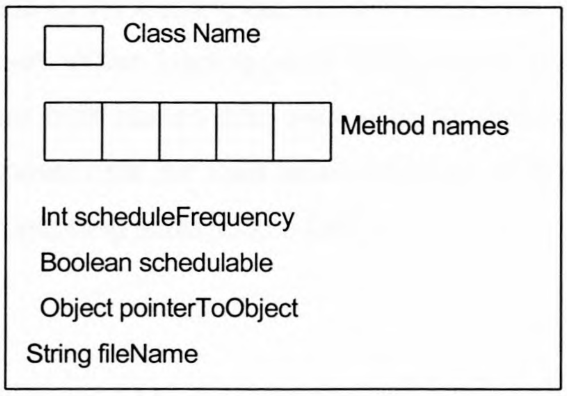


Figure 3.5: Block description of the *classWithMethods object*

3.1.4 Task Loading Objects

The *TaskLoading object* (i.e. *"loadTasks.java"*) is used to load the tasks to the scheduler. Before the scheduler starts to execute, tasks that are supposed to be scheduled are loaded into the scheduler using this object. The *TaskLoading object* takes a file name string as a parameter. The string file name is the name of the text file to read tasks names from. When executing, the *TaskLoading* object reads the name of the task to schedule and attempts to load the task to the list of schedulable tasks. Loaded with each task is its schedule information. Before attempting to load a task, the *TaskLoading object* checks that the class it has to load exists and if it does not exist, then the classNotFoundException is caught. However if the file is there, then the class loader will load it to the LL.

There are two files that the *TaskLoading object* attempts to load tasks from in this project *(i.e. "extraTasks.txt"* and *"TasksToLoad.txt")*. *"ExtraTasks.txt"* contains information about tasks created when the scheduler was already running. When scheduled, the *TaskLoading object* uses the *file-Reader object* to open the text file *"ExtraTasks.txt"* and get the names of extra tasks and the *TaskLoading object* loads those tasks to the LL and then the file *"ExtraTasks.txt"* is closed. The *"ExtraTasks.txt"* file can be edited when not in use (i.e. when not in use, this file can be edited and new tasks names can be edited in this file). The next time the new tasks are supposed to be loaded, *"extraTasks.txt"* will contain new information. If there are no extra tasks to add to the existing list, the file “ExtraTask.txt” will be empty and there will be no task to load. When loading tasks when the scheduler starts, the *TaskLoading object* will get the task names and their schedule information from *“TasksToLoad.txt”*. Figure 3.6 shows how the load tasks object executes when requested to load a task to the LL containing schedulable tasks.

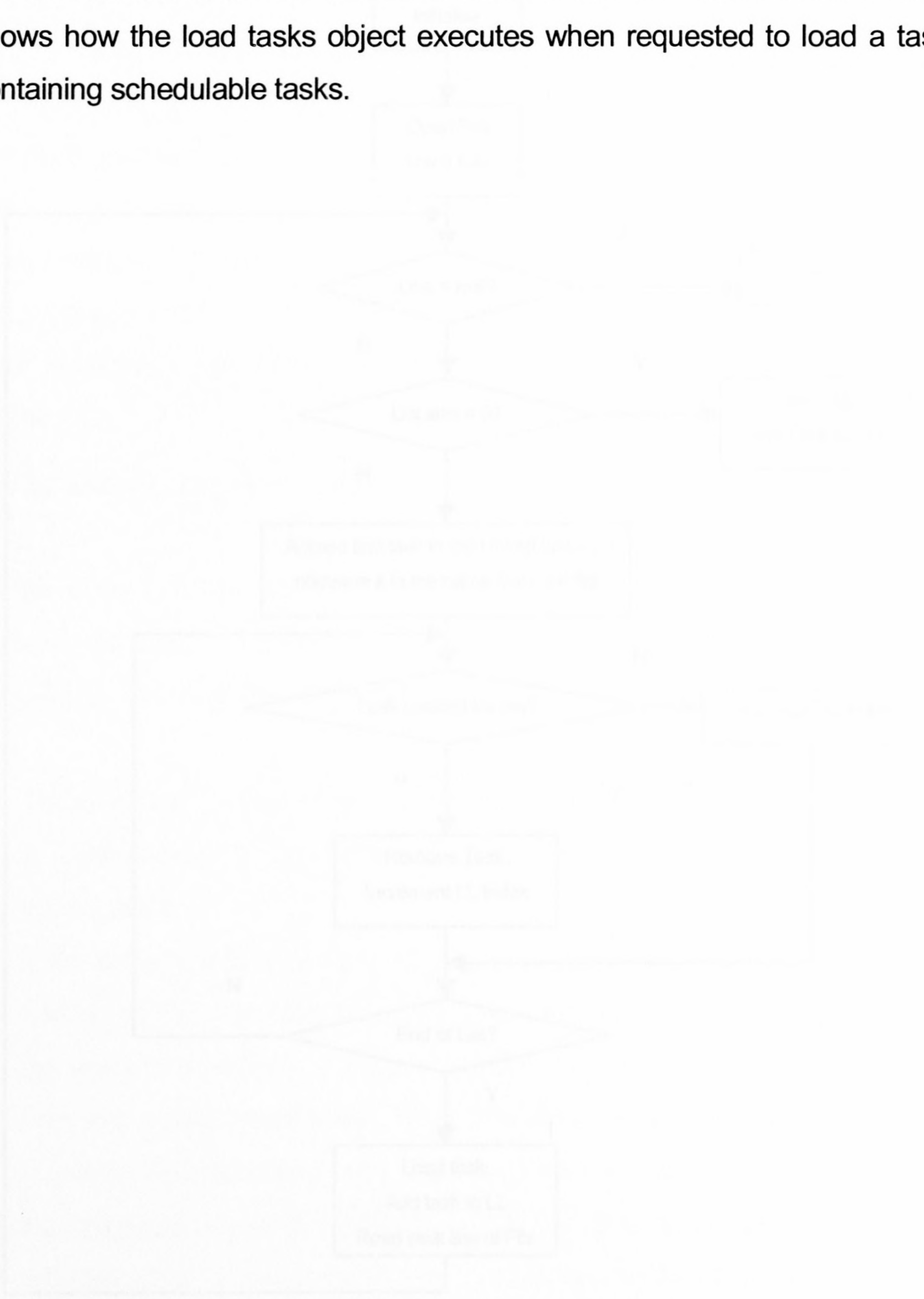


Figure 3.6: Task Loading Object flow diagram

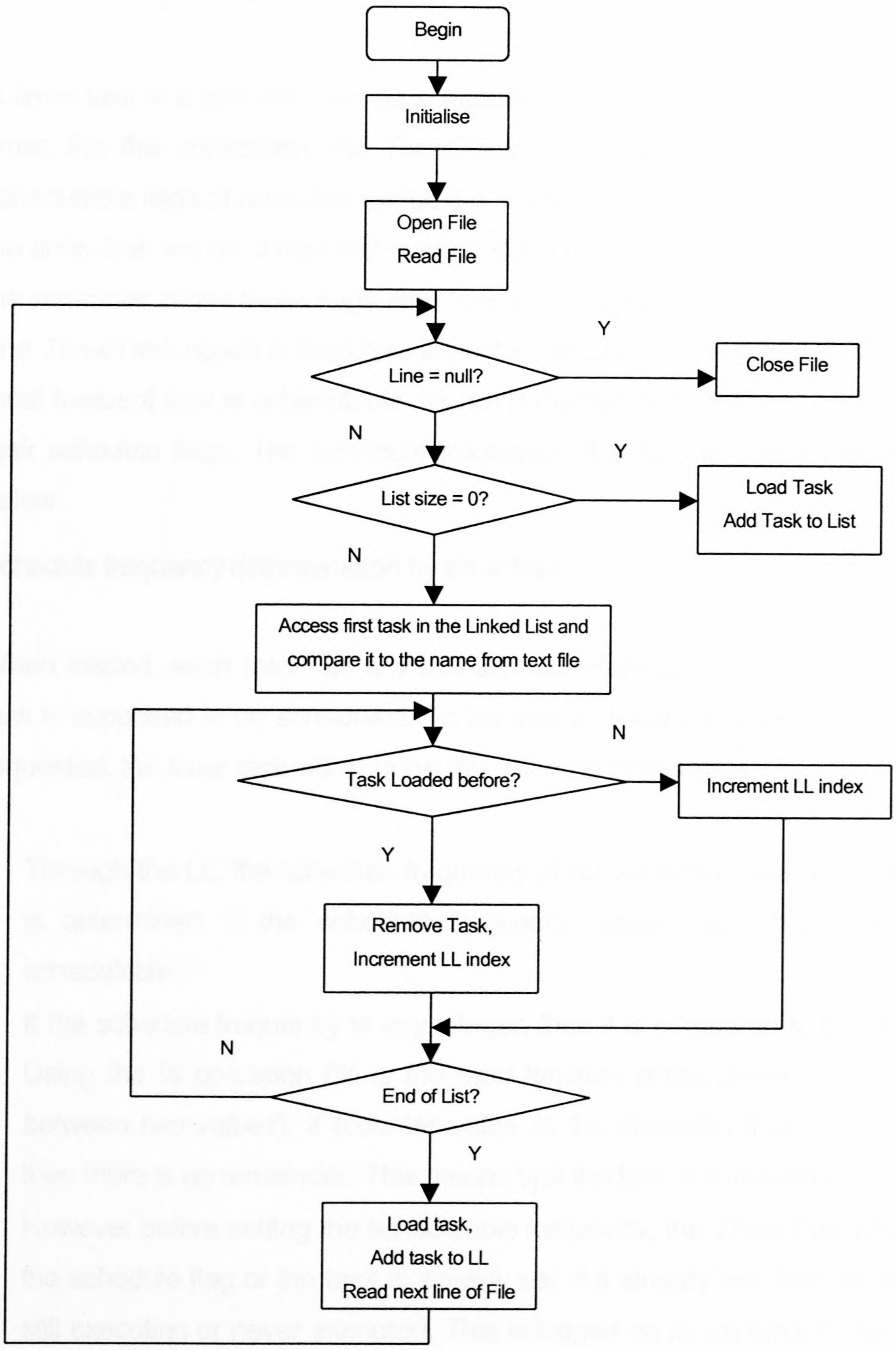


Figure 3.6: Task Loading Object flow diagram

3.1.5 TimerTask Object

A timer task is a task that can be scheduled for one-time or repeated execution by a timer. For this application, the *TimerTask object* (i.e. *"RemindedTasks"*) sets tasks' schedulable flags at repeated cycles (i.e. if a task is schedulable every 1 second, then the timer-task will be scheduled every 1 second to set the task' schedulable flag) and the scheduler clears those flags after they are scheduled.
The *TimerTask object* is therefore important and has to be scheduled every time the most frequent task is schedulable. Its *run ()* method sets tasks schedulable by setting their schedule flags. The schedule frequency of a task is determined as described below.

Schedule frequency determination by timer task

When loaded, each task has a value schedulePeriod, which indicates how often a task is supposed to be scheduled. To be able to ensure that tasks are scheduled as requested, the timer task will perform the following procedures.

- Through the LL, the schedule frequency of each loaded *class With Method object* is determined. If the schedule frequency equals zero, then the task is not schedulable.
- If the schedule frequency is any integer, then it is compared to the counter value. Using the % operation (*% is the mod function which determines the remainder between two values*), if (counter value % the schedule frequency) equals zero, then there is no remainder. This means that the task is schedulable.
- However before setting the schedulable frequency, the *TimerTask object* checks if the schedule flag of the task is already set. If it already set, then either the task is still executing or never executed. This is logged on as an error in the error log file "*Error.text*" and specifies the time the task was previously supposed to be scheduled.
- If the flag is not set, then the schedulable flag is set.
- If there is no remainder in the operation, (*counter % schedule frequency*), then the counter value will be increased by a value equivalent to a second.

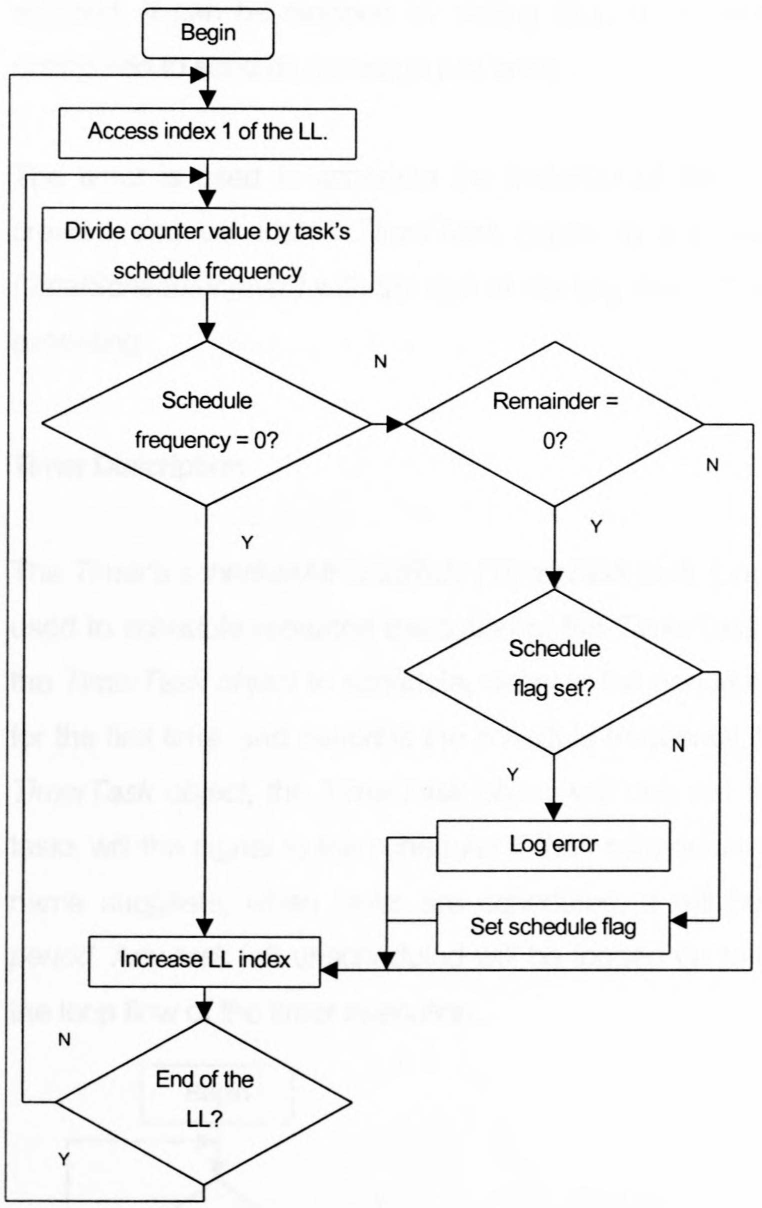


Figure 3.7: The flow diagram for the TimerTask

3.1.6 Timer

The *Timer* (i.e. *an instance of Java's Timer class*) causes an action to occur at a predefined rate. Each Timer has a list of *ActionListeners* and a *delay* i.e. the time between *actionPerformed ()* calls. When *delay* milliseconds have passed, a Timer sends the *actionPerformed ()* message to its listeners. This cycle repeats until stopped. It can be stopped by calling *stop ()*, or halts immediately if the Timer is configured to send its message just once.

The timer is used to schedule the instance of the *TimerTask object*. The *timer* is created and schedules *TimerTask object* in the constructor of the main thread *("finalScheduler.java")* with the aim of starting the timer as soon as the program starts executing.

Timer Description

The *Timer's scheduleAtFixedRate (TimerTask task, long delay, long period)* method is used to schedule repeated execution of the *TimerTask object*. In this method *task* is the *TimerTask object* to schedule, *delay* is the period delay before task is scheduled for the first time, and *period* is the schedule frequency. When the timer schedules the *TimerTask object*, the *TimerTask object* will only set the task schedulable flag. The tasks will the signal to the scheduler if their schedulable flags are set. As the method name suggests, when tasks are scheduled, it will be at fixed rate determined by *period.* Any task left unscheduled will be logged on to the error log file. Figure 3.8 is the loop flow of the timer execution.

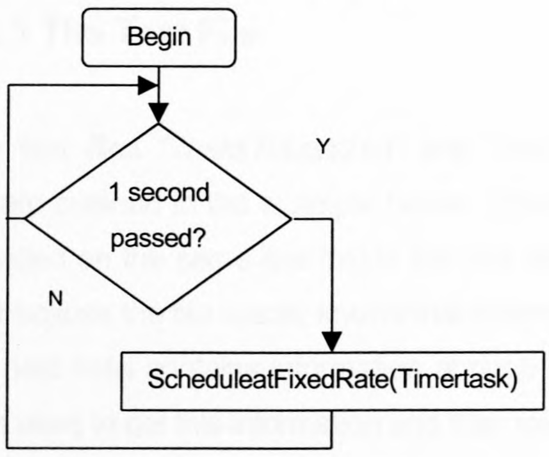


Figure 3.8: Timer flow diagram

3.1.7 Dynamic Linking

Java classes are linked together as they are needed during runtime. The Java language knows where it should look for classes that need to be linked while a Java program runs. Dynamic linking makes it easier to keep Java programs up-to-date since the latest version will always be used [20]. Using Java's dynamic LL, task LL was being able to be dynamically extended. Therefore using a user defined class loader, the application can be loaded and dynamically link to classes and interfaces that were unknown or did not even exist when the application was compiled.

3.2 Software Implementation

The aim of the research was to lay a foundation for the possible implementation in embedded environment. Since the scheduler was implemented on a PC, it is very important to specify how the scheduler was implemented. The ultimate aim is to implement the scheduler on SUNSAT 2 and this subsection investigates whether it will be possible to implement the scheduler on a satellite (i.e. SUNSAT 2).

3.2.1 The Text File

The text files *"TasksToLoad.txt" and "nextTasks.txt*" were designed to follow the pattern outlined in the example below. Information about the task to be scheduled is provided on the same line inside the text file. This makes it easier when reading the file because the file reader knows that information in one line belongs to one task and the next lines contains information about the other tasks. The *readLine ()* method is then used to get this information and then loaded for scheduling.

Example

doubleClass:incDouble:10:true

IntClass:incInt:1:true

Figure 3.9: Example of the fields of text file

Where strings *doubleClass* and *intClass*, are the names of the two classes/tasks to be loaded. Strings *incDouble* and *incInt* are the method names to schedule respectively. Strings 10 and 1 are casted as integers and are schedule frequency of tasks in seconds respectively. String true is then casted as a Boolean, which will represent the schedule flag of each task. The colons (:) are the delimiters, used to separate between strings.

3.2.2 Data Object

During scheduler execution, tasks will read data from sensors and use information gathered in ADCS control algorithms for control of the satellite. *Data object* represents that data and it is a pool of data of object type. *Data object* creates empty linked lists. For every object type, there is a different LL type. For instance a LL for Double types, Integer types and Floats types. As tasks are executing, they might all want to store data. If however many tasks are all writing to the same LL, it might be difficult to determine where did a particular task stored its data. Hence the idea of having a separate LL for each object type.

There are two types of LL for each type. The new LL used mainly to temporarily store data during task execution (before the data object LL is updated) and the old LL is used to store data when the data object is updated. Data from new LL is usually stored in the old LL as it might be used by other tasks in the future and also because information stored has to be used to control the satellite to a particular orientation. This object has one method *update ()* which replaces the old LL with new LL when scheduled. When updating the old LL (i.e. new LL becomes old LL), the *get (int index)* method is used to return the element at the specified positions in this new LL. Then the linked list's *set (int index, Object element)* method is used to replace previously stored data in the old LL by replacing it with this new data. Data Object is scheduled every second because all the tasks use it. Since this object will be scheduled every second, the data pool will be updated every second.

3.2.3 AddLinkedList Object

AddLinkedList object (i.e. *"addLinkedList.java"*) adds elements to the linked lists created in the *data object*. When scheduling *data object*, the aim is to update the data pool contained inside. Therefore *AddLinkedList object* initializes the LL by adding data to them.

AddLinkedList object description

AddLinkedList object has the method *upDateLinkedList ()* and takes *data object* as a parameter. Before attempting to add any data to the linked lists, to ensure that the linked lists are empty, the list are cleared off all the elements it contains using the *clear* method. The *add (int index, Object element)* is then used to insert the specified element (i.e. data) at the specified position in the LL. Elements currently at that position (if any) and any subsequent elements will be shifted to the right. (i.e. adds one to their indices)

3.2.4 Task Files

Task file are those classes scheduled by the scheduler. Task files take *data object* as an argument because all data is stored in *the data object*. This *data object* is stored in index one of the LL, therefore to access data from the LL, the *get (int index)* method is used to access it. The *invoke (Object object, Object [] args)* then schedules the task when the task is due for scheduling. The Object *object* is the object to schedule (i.e. the task). Object array args is the parameters used by the scheduled object (i.e. data object as it is the argument of many tasks).

The LL stores object types and to use the data contained in the LL, the object type has to be unwrapped to primitive type. To store data in the in the LL after operation on primitive types, they have to be wrapped back as object types. Figure 3.7 illustrates what is discussed above.

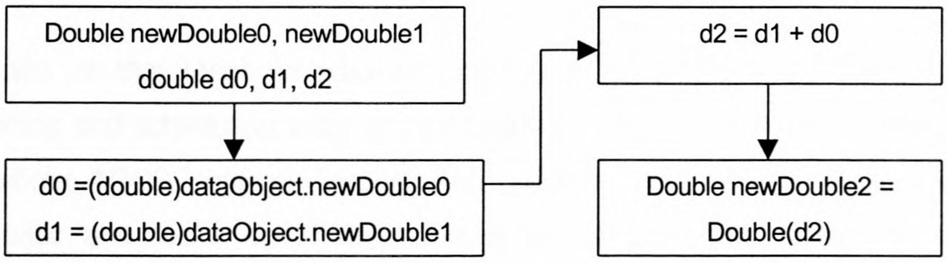

Double newDouble0, newDouble1
double d0, d1, d2
d0 =(double)dataObject.newDouble0
d1 = (double)dataObject.newDouble1
d2 = d1 + d0
Double newDouble2 =
Double(d2)

Figure 3.10: Block diagram of using parameters in a task file

3.2.5 Summary

This chapter investigated the structure of the scheduler software and the scheduler implementation. The main focus was to discuss the structure of each object used and the structure of the objects used for implementation. The scheduler has been designed to have two threads (i.e. scheduler thread and the watchdog thread) executing in parallel. The main thread being the scheduler with the watchdog thread ensuring that the main scheduler is executing to the required level of performance. The next chapter gives the evaluation and testing of scheduler performance.

CHAPTER 4: EVALUATION AND TESTING

The aim of this research was to implement an effective scheduler capable of uploading and scheduling extra control tasks and algorithms without having to upload the whole ADCS software image. This section of the thesis evaluates the new scheduler and looks at whether the objective was met. It also evaluates the advantages of the new scheduler against the current scheduler. The features that were improved from the CSS are also looked at in this chapter.

4.1 Comparison to the current scheduler

The approach to the design new scheduler is similar to the current scheduler with the exception of implementing the extra requirements (e.g. Uploading of extra tasks) specified for the new scheduler. In the current scheduler, when there are new tasks available, the whole software has to be uploaded because the task algorithms are embedded on the scheduler algorithm. The new scheduler exploits Java's class loader to ensure that new tasks can be loaded on the schedulable list when the scheduler is already running. This ensures that the scheduler is not stopped when new tasks are loaded to the scheduler.

In the new scheduler, the scheduler does not need to know anything about the task to be loaded whilst in the current scheduler, the tasks to schedule are known before hand as they are embedded in the scheduler software. In the new scheduler, only the extra tasks needs to be compiled, upload that task and update the text file. Therefore this option gives an advantage of being able to create tasks at anytime when the scheduler is already running. This is made possible by Java 's linking model, where you are able to dynamically link and extend Java applications.

The other advantage of the new scheduler is its ability to schedule tasks at any time frequency. The current scheduler is only able to schedule tasks at 1 second and 10 seconds' intervals. When the scheduler executes, the timer task object checks for each task's schedule frequency and sets the task's schedule flag if it is due for scheduling.

4.2 Scheduling Extra tasks

Scheduling extra tasks in the new scheduler is one of the notable fixtures. When scheduling extra tasks, (i.e. those created when the scheduler is already running) there is no need to stop the scheduler because the scheduler is designed in such a way that it checks for those tasks created late when the scheduler was already running. This is achieved by having a task (called newTasks and is an instance of *loadClass object*) designed especially to load extra tasks. When scheduled, newTasks object will check if there are any new tasks to load. The tasks specified by the newTasks object will be added in the LL of all the other tasks ready to be scheduled.

4.3 Code sharing and code re-use

Using Java as object orientated language, there is an advantage of re-using codes and sharing of codes. An example of a re-used code in this project is utilizing the *load tasks object*. Since there is no difference in the structures of the tasks, *load tasks object* is used to load both tasks created when the scheduler starts running and to load the new tasks. Instead of writing two similar classes with similar purpose, one class is created and is instantiated twice when it is used. This makes the code portable and easy to debug.

On code sharing, Java offers an advantage of having an object, which can be shared amongst other objects. An instance where a code is shared is data object. Since data object is used by most tasks to store data and read data from, it has to be scheduled every time a task requires it to be scheduled. At times, a lot of tasks are schedulable at the same time, and with most tasks requiring data object as a parameter. The tasks are able to access *data object* (either read data from it or store data in it) at the same time (i.e. the time they are all scheduled).

4.4 Error detection and logging

While the main thread is being executed, one task may access the processor longer than the others, or even worse it might access the processor indefinitely. This could be due to a number of reasons. The tasks may be running an infinite loop or a task executes longer than the scheduler accesses the processor. To guard against this the watchdog thread is designed to ensure that tasks are executed in their allocated run time.

The main loop executes every second and if the main loop takes longer than 1.2-seconds then the watchdog thread will kill/reset the main thread and create a new main thread. All the tasks that are skipped are logged in the *"Error.txt"*. In the *"Error.txt"* file, the name of the task and the time the task was supposed to be scheduled are logged.

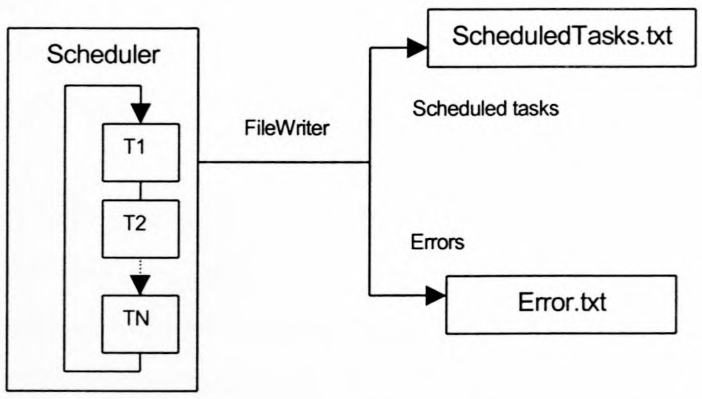


Figure 4.1: Performance and Error logging evaluation

4.5 Summary

From this chapter, the information gathered is that the new scheduler is able to upload and schedule extra control tasks and algorithms without having to upload the whole software image. There is no need to stop the scheduler when adding new tasks to the scheduler. Also compared to the current scheduler, the new scheduler has many advantages as discussed in the above sections. The next chapter focuses on the conclusions reached about the new scheduler and makes a few recommendations for possible future researches.

CHAPTER 5: CONCLUSIONS

This chapter gives conclusions regarding the overall implementation and performance of the new scheduler and gives the shortcomings and recommendations for possible future research. The previous chapter looked at the performance of the new scheduler, and this chapter starts by looking at conclusions on the performance of the new scheduler and gives a general conclusion on the implementation of the new scheduler.

5.1 Performance analysis of the new scheduler

Performance analysis of the new scheduler was mainly based on error reporting of every task that is not scheduled. All the tasks that are skipped unscheduled are logged on to the *"Error.txt"* file. *"Error.txt"* contains the names of the skipped tasks and the time it was supposed to be scheduled. Those that are scheduled are also logged on in the *"ScheduledTasks.txt"*. This file contains the name and the time the task was scheduled. Information from two these text files gives a slight indication of how reliable the new scheduler is.

The watchdog thread is another error detecting evaluation mechanism that was used. Every time the main thread is reset, an error is reported *"Error.txt"*. Knowing how often the main thread is reset gives an idea with regard to how the scheduler performs. If the main thread is reset often, then it means that the scheduler is not performing well and if the main thread is not reset often, then the scheduler is performing well. Every time the main thread is reset the time it is reset is logged in the error file.

5.2 Short comings on the new scheduler

The run-to-completion scheduling policy gives the task access to the processor until the task has finished running. Since a task can run into an infinite loop, the scheduler then risks running one task forever without passing the processor to other tasks. By killing the thread every time the main thread executes longer than expected, we are able to guard against the risk of having one task holding on to the processor forever.

Also the scheduler was designed to have certain tasks at certain positions in the LL only (data object at index 1 and extraTasks at index 2 in the LL). If these tasks positions are changed (either by adding new tasks in their positions or by loading them at different position at the start of the program execution), then the scheduler risks collapsing. Therefore it is very important to follow certain guidelines (e.g. that data object occupy index 1 of the linked list always and new tasks object at index 2 of the LL) on the implementation of this scheduler for its effective use.

5.3 Recommendations

- The new scheduler was implemented on a PC. It would therefore be interesting to implement the new scheduler in an embedded environment on either ADCS 's T800 or OBC's 386 processor.

- Make the application be able to execute on simulation mode. This is the accelerated, faster than real-time mode, execution of the scheduler. It has a simulated sensor data input for accelerated execution of the algorithms.

5.4 Summary

In summarising the new scheduler, the conclusion is that the objectives of the project were met. Since the project involved implementing an effective scheduler capable of uploading and scheduling extra tasks without having to upload the whole software image, the new scheduler does meet all the specifications (i.e. Uploading and scheduling extra control tasks and algorithms). Compared to the current scheduler, the new scheduler gives some new advantages. These advantages include having a scheduler running and scheduling tasks, which were not known when the scheduler started to execute. With advantages like these, the new scheduler can be implemented in SUNSAT 2, and it would be advantageous for satellite orientation determination and control.

REFERENCES

[1] Tanenbaum A.S. and Woodhull A.S. (1997) *"Operating systems: Design and implementation 2nd edition"*, Prentice-Hall Int, New Jersey page 83-84

[2] ACP Software implementation (2001), Programmer W.H. Steyn and documented by J.A.A. Engelbrecht. Stellenbosch University, SA.

[3] ACP Software implementation (2001), Programmer W.H. Steyn and documented by J.A.A. Engelbrecht. Stellenbosch University, SA.

[4] B. Venners (1999) *"Inside the Java2 virtual machine second edition"*, McGraw Hill, US, page 6

[5] Borland JBuilder2 Quick Start, 1998 page 19-5

[6] Li Gong (1999) *"Inside Java 2 Platform Security Architecture, API Design, and Implementation"* Sun Microsystems, Addison Wesley, Palo Alto CA, page 73

[7] B. Venners (1999) *"Inside the Java2 virtual machine second edition"*, McGraw Hill, US, page 238

[8] B. Venners (1999) *"Inside the Java2 virtual machine second edition"*, McGraw Hill, US, page 45

[9] B. Venners (1999) *"Inside the Java2 virtual machine second edition"*, McGraw Hill, US, pages 59-60

[10] Li Gong (1999) *"Inside Java 2 Platform Security Architecture, API Design, and Implementation"* Sun Microsystems, Addison Wesley, Palo Alto CA, pages 76-77

[11] Li Gong (1999) *"Inside Java 2 Platform Security Architecture, API Design, and Implementation"* Sun Microsystems, Addison Wesley, Palo Alto CA, pages 73

[12] B. Venners (1999) *"Inside the Java2 virtual machine second edition"*, McGraw Hill, US, page 9

[13] B. Venners (1999) *"Inside the Java2 virtual machine second edition"*, McGraw Hill, US, page 270

[14] Wertz J.R. and Larson W.J. (1999) "*Space Mission Analysis and*

Design 3rd edition", Kluwer Academic Publishers, Space Technology Library, Torrance CA, page 354

[15] Wertz J.R. (1999) *"Spacecraft Attitude Determination and Control"*, Kluwer Academic Publishers, Astrophysics and Space Science Library, Torrance CA, page 354

[16] Maral G. and Bousquet M. (1998) *"Satellite Communications Systems, Systems, Techniques and Technology 3rd edition"*, John Wiley & Sons, Toulouse France, page 535

[17] Engelbrecht J.A.A (1999) *"A HIL simulation for the SUNSAT ADCS"*, Masters Thesis, University of Stellenbosch, SA, page 24

[18] Wertz J.R. and Larson W.J. (1999) "*Space Mission Analysis and Design 3rd edition*", Kluwer Academic Publishers, Space Technology Library, Torrance CA, page 369

[19] Engelbrecht J.A.A (1999) *"A HIL simulation for the SUNSAT ADCS"*, Masters Thesis, University of Stellenbosch, SA, page 23

[20] Wertz J.R. (1999) *"Spacecraft Attitude Determination and Control"*, Kluwer Academic Publishers, Astrophysics and Space Science Library, Torrance CA, page 10

[21] Steenkamp N.L. (1999) *"Development of the On Board Computer Fight Software for SUNSAT I"*, Masters Thesis, University of Stellenbosch, SA, page 13

[22] Wertz J.R. and Larson W.J. (1999) "*Space Mission Analysis and Design 3rd edition*", Kluwer Academic Publishers, Space Technology Library, Torrance CA, page 371

[23] Engelbrecht J.A.A (1999) *"A HIL simulation for the SUNSAT ADCS"*, Masters Thesis, University of Stellenbosch, SA, page 22

[24] Wertz J.R. and Larson W.J. (1999) "*Space Mission Analysis and Design 3rd edition*", Kluwer Academic Publishers, Space Technology Library, Torrance CA, page 374

[25] Maral G. and Bousquet M. (1998) *"Satellite Communications Systems, Systems, Techniques and Technology 3rd edition"*, John Wiley & Sons, Toulouse France, page 530

[26] Engelbrecht J.A.A (1999) *"A HIL simulation for the SUNSAT ADCS"*, Masters Thesis, University of Stellenbosch, SA, page 23

[27] ACP Software implementation (2001), Programmer W.H. Steyn and documented by J.A.A. Engelbrecht. Stellenbosch University, SA.

APPENDICES

Appendix A: Current SUNSAT Scheduler Software execution

This section investigates the execution of the previous software on SUNSAT. In summarising, the main execution loop executes until it reaches the semaphore and then waits for one of the three threads to signal the semaphore. When the ACP is executing on its normal real time mode, the ICP will send a data packet to the ACP once every second, causing the ICP thread to signal the semaphore. When the ACP is running in simulation mode, the OBC will send simulated sensor packets at a much higher rate than once per second, causing the OBC thread to signal the semaphore. If both the ICP thread and the OBC thread fail to signal the semaphore, the watchdog timer thread will time out after 1.2 seconds and signal the semaphore, allowing the main loop to resume execution.

The Main Execution Loop

A flowchart, which describes the execution of the main loop of the ACP software, is shown in figure 7.1. The main execution loop for the current ACP software is extracted from ACP software.

Initialization

1. Set the time since epoch is set to zero.
2. Initialize the semaphore.
3. Start the watchdog timer thread.

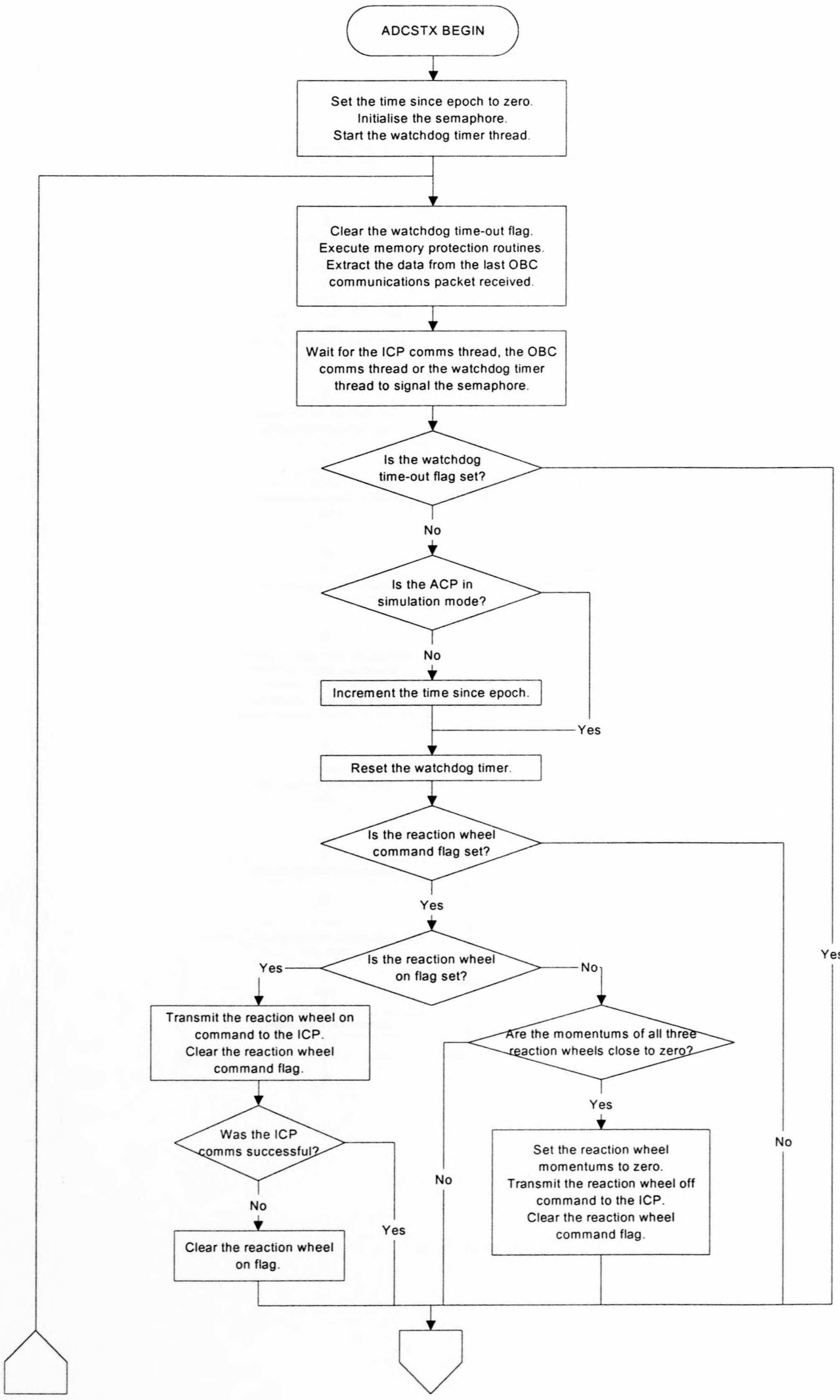
Execution

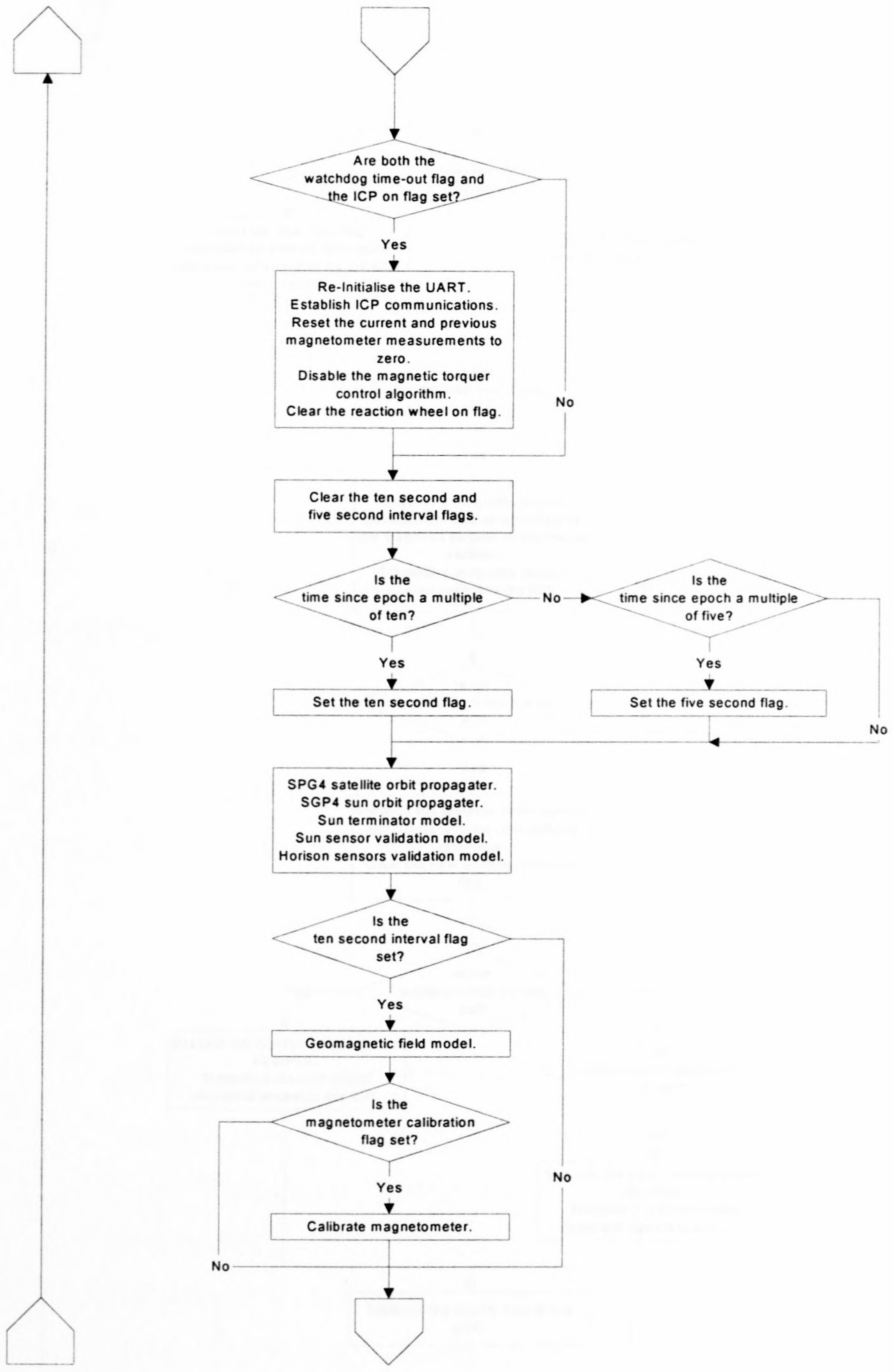
1. Clear the watchdog time-out flag.
2. Check and correct the protected variables in GLOBAL.DEF, OBCTX.DEF and

RWHEELTX.DEF.

3. Extract the data from the last OBC packet, which was received by the OBC thread.
4. Wait for the semaphore to be signaled by the ICP thread, the OBC thread or the watchdog timer thread.
5. If a watchdog time-out did not occur, then:
    a) If the ACP is running in simulation mode, increment the time since epoch.
    b) Reset the watchdog timer.
    c) If the reaction wheels should be commanded, then:
        i) If the reaction wheels should be on, then:
            (1) Transmit the reaction wheel on command to the ICP.
            (2) Clear the reaction wheel command flag.
            (3) If the ICP packet was not transmitted successfully, clear the reaction wheel on flag.
        ii) Otherwise, if the reaction wheels should be off, then:
            (1) If the reaction wheel momentums are near to zero, then:
                (a) Set the momentums to zero.
                (b) Transmit the reaction wheel off command to the ICP.
                (c) Clear the reaction wheel command flag.
6. If an ICP time out occurred (both the time out flag and the ICP on flag are set):
    a) Re-initialize the UART.
    b) Re-establish communications with the ICP.
    c) Reset the current and previous magnetometer measurements to zero.
    d) Disable the magnetic torquer controllers.
    e) Clear the reaction wheel on flag.
7. Clear the ten-second and five-second interval flags. If the time since epoch is a multiple of ten, set the ten second flag, otherwise if the time since epoch is a multiple of five, set the five second flag. These flags will be used by tasks, which should be scheduled at ten-second or five-second intervals.
8. Execute the SPG4 orbit satellite propagator.
9. Execute the SGP4 sun orbit propagator.
10. Execute the sun terminator model.

11. Execute the sun sensor validation model.
12. Execute the horizon sensors validation model.
13. If the ten second interval flag is set, then:
    a) Execute the geomagnetic field model (the IGRF model).
    b) If the calibrate magnetometer flag is set, then execute the magnetometer calibration algorithm.
14. If the new filter flag is set, then clear the new filter flag and initialize a new attitude estimation algorithm as specified by the filter mode variable.
15. Otherwise, execute the current attitude estimation algorithm as specified by the filter mode variable.
16. If the ten second flag is set, then execute the current magnetic torquer control algorithm as specified by the magnetic control variable and transmit the magnetic torquer control packet to the ICP.
17. If the new attitude reference flag is set, then initialize the reaction wheel control algorithms with the new attitude reference and clear the new attitude reference flag.
18. If the reaction wheel on flag is set, then execute the current reaction wheel control algorithm and transmit the reaction wheel control packet to the ICP.
19. Otherwise (if the reaction wheel on flag is clear):
    a) If the reaction wheel command flag is set, execute the stop reaction wheels algorithm and transmit the reaction wheel control packet to the ICP.
    b) Otherwise, if the reaction wheel command flag is clear, then set the reaction wheel control torques to zero.
20. Transmit the ADCS data to the OBC via the links.
21. Repeat from step 1.


ADCSTX BEGIN
Set the time since epoch to zero. Initialise the semaphore. Start the watchdog timer thread.
Clear the watchdog time-out flag. Execute memory protection routines. Extract the data from the last OBC communications packet received.
Wait for the ICP comms thread, the OBC comms thread or the watchdog timer thread to signal the semaphore.
Is the watchdog time-out flag set?
No
Is the ACP in simulation mode?
No
Increment the time since epoch.
Yes
Reset the watchdog timer.
Is the reaction wheel command flag set?
Yes
Is the reaction wheel on flag set?
Yes
No
Transmit the reaction wheel on command to the ICP. Clear the reaction wheel command flag.
Are the momentums of all three reaction wheels close to zero?
Yes
Was the ICP comms successful?
No
Yes
Clear the reaction wheel on flag.
Set the reaction wheel momentums to zero. Transmit the reaction wheel off command to the ICP. Clear the reaction wheel command flag.
No
No
Yes


Are both the watchdog time-out flag and the ICP on flag set?
Yes
No
Re-Initialise the UART.
Establish ICP communications.
Reset the current and previous magnetometer measurements to zero.
Disable the magnetic torquer control algorithm.
Clear the reaction wheel on flag.
Clear the ten second and five second interval flags.
Is the time since epoch a multiple of ten?
No
Is the time since epoch a multiple of five?
Yes
Yes
Set the ten second flag.
Set the five second flag.
No
SPG4 satellite orbit propagater.
SGP4 sun orbit propagater.
Sun terminator model.
Sun sensor validation model.
Horison sensors validation model.
Is the ten second interval flag set?
Yes
Geomagnetic field model.
Is the magnetometer calibration flag set?
Yes
No
Calibrate magnetometer.
No

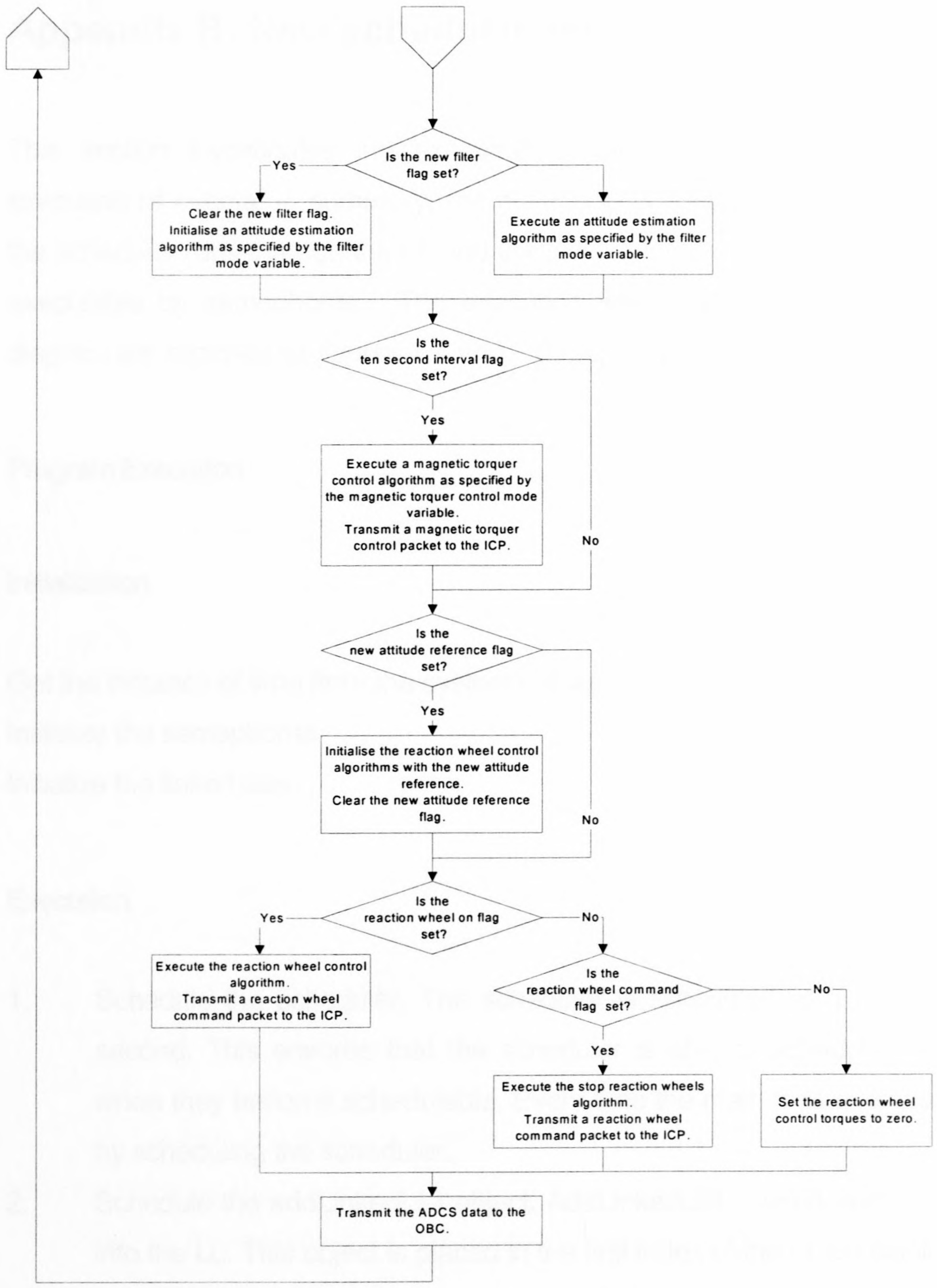

Is the new filter flag set?
Yes
Clear the new filter flag. Initialise an attitude estimation algorithm as specified by the filter mode variable.
Execute an attitude estimation algorithm as specified by the filter mode variable.
Is the ten second interval flag set?
Yes
Execute a magnetic torquer control algorithm as specified by the magnetic torquer control mode variable. Transmit a magnetic torquer control packet to the ICP.
No
Is the new attitude reference flag set?
Yes
Initialise the reaction wheel control algorithms with the new attitude reference. Clear the new attitude reference flag.
No
Is the reaction wheel on flag set?
Yes
No
Execute the reaction wheel control algorithm. Transmit a reaction wheel command packet to the ICP.
Is the reaction wheel command flag set?
No
Yes
Execute the stop reaction wheels algorithm. Transmit a reaction wheel command packet to the ICP.
Set the reaction wheel control torques to zero.
Transmit the ADCS data to the OBC.

Figure A.1: A flow chart of the ACP main execution loop

Appendix B: New scheduler execution flow chart

This section investigates the execution of the new scheduler and the block execution of events. In summary, the main loop is executed every second, where the scheduler runs through the LL and the tasks that are stored in this LL will be set executable by semaphores. The execution flow chart and the execution block diagram are attached as figures 7.4 and 7.5 respectively.

Program Execution

Initialization

Get the instance of time from the system's clock.
Initialize the semaphores
Initialize the linked lists

Execution

1. Schedule the scheduler. The scheduler is scheduled at a fixed rate every second. This ensures that the scheduler is able to schedule all the tasks when they become schedulable. Every time the main loop executes, it starts by scheduling the scheduler.
2. Schedule the addLinkedList object. AddLinkedList is used to initialize values into the LL. This object is placed in the first index of the LL containing objects to be scheduled. This object is scheduled only once and that is when the main loop executes for the first time. The main object of this object is to initialize the first values into the linked lists. Once there are values into the linked lists, tasks will use those values and will also store new values.
3. Then schedule data object. Data object is used by many tasks as an argument so as to store data to it and read data from it. This object is schedulable every second so as to ensure that it is scheduled with all the

tasks that are schedulable as frequent as possible.

4. New tasks task is placed at index two of the LL. It is used to add new tasks to the LL and is schedulable at very slow rate. Once scheduled, it loads new tasks to the LL and that link list 's objects will be added to the existing LL with their respective information.
5. All the other tasks will be scheduled as follows:

   Check if 10 seconds has passed

   If 10 seconds has passed

   a. Begin by scheduling dataObject where the method *upDate ()* is executed. In this method, the new data from operations on tasks is stored as old data.
   b. Then execute the tasks that are schedulable every 10 seconds. Before executing any task, its scheduleFlag is checked to determine if it is still running or there is any error that occurred during its previous schedule. Errors are logged in the error log file.
   c. Then the next execution will be the 1-second tasks. They will always be scheduled when the loop is executed because their schedule frequency will always coincide with all the other tasks. Also their scheduleFlag is checked to determine their state from their previous schedule and errors are also logged in the error log file

Else if 1 second has elapsed, execute the 1-second tasks

   d. Repeat steps a, b and c of the above operations and increase the time counter every time.

Else if *n* seconds has elapsed, execute the *n*-seconds tasks

   e. Repeat steps a, b and c of the above execution and increase the time counter every time

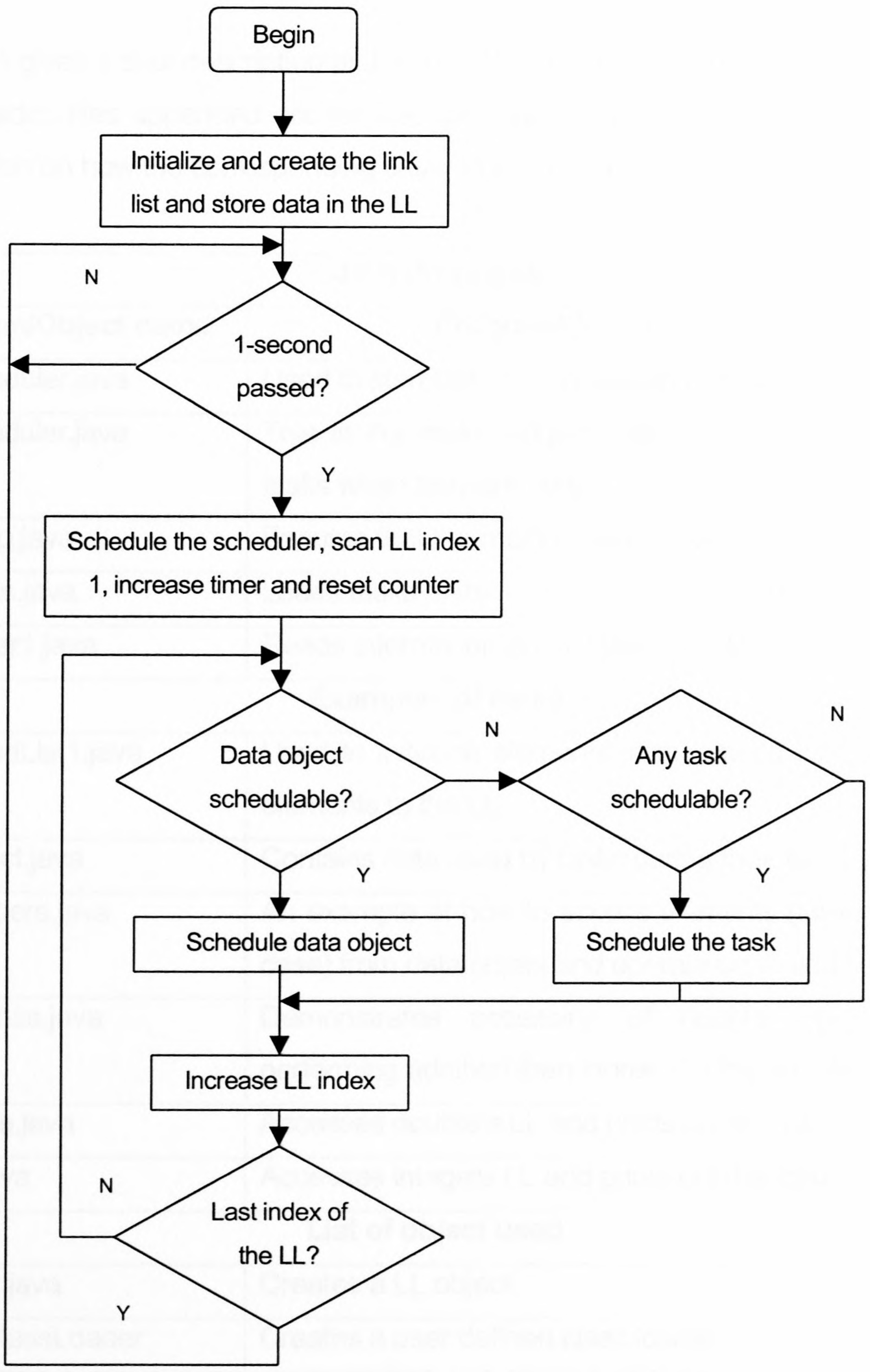


Figure B.1: New scheduler Execution flow chart

Appendix C: Javadoc and Java Code

Table C.1 gives a brief description of Javadoc files that are appended after table 7.1. The Javadoc files appended are for files used during the project and give a brief explanation on how the corresponding Java programs execute.

Java Programs	
Program/Object name	**Program/Object description**
RunScheduler.java	Used to start the main thread and the watchdog thread.
finalScheduler.java	This is the main program, which loads and schedules tasks when they are schedulable.
watchDog.java	Ensures that the main thread is executing well
loadTasks.java	Loads tasks to the LL of schedulable tasks.
PrintWriter1.java	Reads information about tasks from task files.
Examples of tasks	
AddLinkedList1.java	Used to initialize elements in the linked lists by adding elements to the LL.
dataObject.java	Contains data used by tasks during their execution.
addNumbers.java	An example of how to access elements (integers in this case) from data object and operate on it (addition here).
doubleClass.java	Demonstrates accessing of double type LL and performing addition then stores it in the doubles LL.
getDouble.java	Accesses double's LL and prints out the value.
getInt2.java	Accesses integers LL and prints out the value.
List of object used	
linkedList.java	Creates a LL object.
myOwnClassLoader	Creates a user defined class loader.
CounterObject.java	Has a public field counter of int type used by the two threads. The counter is reset by the main thread and incremented by the watchdog when the main is executing well.

classWithMethods.java	Contains information about the tasks.

Table C.1: Description of files in the appended Javadoc

Class RunScheduler

```
java.lang.Object
  |
  +--RunScheduler
```

public class **RunScheduler**
extends java.lang.Object

Starts the main thread (i.e. finalScheduler.java) and the watchdog thread (i.e. watchDog.java) and Executes the two thread's run method.

Constructor Summary

`RunScheduler()`

Method Summary

`static void`	`main(java.lang.String[] args)` Creates a new instance of the schedule object.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

RunScheduler

```
public RunScheduler()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
                 throws java.lang.Exception
```

Creates a new instance of the schedule object. The schedule object will start the two threads (finalScheduler and watchDog) and executes their run methods.

Class finalScheduler

```
java.lang.Object
  |
  +--java.lang.Thread
        |
        +--finalScheduler
```

All Implemented Interfaces:

java.lang.Runnable

public class **finalScheduler**
extends java.lang.Thread

Scheduler program which uses a fileReader to read the names of the tasks to load and loadClass method of ClassLoader to load the tasks and the loaded task is scheduled at the specified frequency.All the tasks that have been scheduled will be written in a text file "ScheduledTasks.TXT" specifying the tasks and the time it has been scheduled and the Errors will be logged in a text file "Error.txt" which also specifies the unscheduled tasks and the time they were supposed to be scheduled

Field Summary

`int`	**`counter`** integer value corresponding to the clock'second
`java.lang.Boolean`	**`False`** The Boolean object corresponding to the primitive value false.
`boolean`	**`TerminateFlag`** Terminates the execution of a task when set true
`java.lang.Boolean`	**`True`** The Boolean object corresponding to the primitive value false.

Fields inherited from class java.lang.Thread
`MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY`

Constructor Summary

`finalScheduler``(CounterObject counter)` Tasks are loaded to the linked list and the scheduler is scheduled and the counter value is determined from the counter object.The constructor takes CounterObject as an argument which parses the integer counter value to the scheduler

Method Summary

`void`	**`run`**`()` Run Method : When the terminate flag is not set, then this method will schedule tasks by invoking schedule method
`void`	**`scheduleMethod`**`(newTasks extraTasks)` Method to schedule loaded methods.This method takes newTasks as a parameter to enable it to schedule new tasks.As this method executes, it also resets the counter.If the counter is not reset, then the the watchdog thread will reset the counter after 1.2 seconds.When executing, this method will run through the linked list and check the schedule flag for every task.If set then the scheduler will give the task the processor.The scheduler will check the following conditions:- 1: If the task is at index one of the linked list (i.e data object), then that object takes null when invoked 2: If the flag set signals task at index two, then new tasks will be loaded to the scheduler 3: All the other object will take data object as an argument when scheduled, Then their schedulable flag will be cleared after the task has been scheduled.

Methods inherited from class java.lang.Thread
`activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getContextClassLoader, getName, getPriority, getThreadGroup, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setName, setPriority, sleep, sleep, start, stop, stop, suspend, toString, yield`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Field Detail

TerminateFlag

```
public boolean TerminateFlag
```

Terminates the execution of a task when set true

True

```
public java.lang.Boolean True
```

The Boolean object corresponding to the primitive value false.

False

```
public java.lang.Boolean False
```

The Boolean object corresponding to the primitive value false.

---

counter

```
public int counter
```

integer value corresponding to the clock'second

Constructor Detail

finalScheduler

```
public finalScheduler(CounterObject counter)
```

Tasks are loaded to the linked list and the scheduler is scheduled and the counter value is determined from the counter object.The constructor takes CounterObject as an argument which parses the integer counter value to the scheduler

Method Detail

scheduleMethod

```
public void scheduleMethod(newTasks extraTasks)
```

Method to schedule loaded methods.This method takes newTasks as a parameter to enable it to schedule new tasks.As this method executes, it also resets the counter.If the counter is not reset, then the the watchdog thread will reset the counter after 1.2 seconds.When executing, this method will run through the linked list and check the schedule flag for every task.If set then the scheduler will give the task the processor.The scheduler will check the following conditions:- 1: If the task is at index one of the linked list (i.e data object), then that object takes null when invoked 2: If the flag set signals task at index two, then new tasks will be loaded to the scheduler 3: All the other object will take data object as an argument when scheduled, Then their schedulable flag will be cleared after the task has been scheduled.

---

run

`public void **run**()`

Run Method : When the terminate flag is not set, then this method will schedule tasks by invoking schedule method

Overrides:

run in class `java.lang.Thread`

Class watchDog

```
java.lang.Object
  |
  +--java.lang.Thread
        |
        +--watchDog
```

All Implemented Interfaces:

java.lang.Runnable

public class **watchDog**
extends java.lang.Thread

Used to watch if the scheduler's loop is executed every second.If the loop is not executed after 1.2 second, then this class resets the scheduler.

Fields inherited from class java.lang.Thread
`MAX_PRIORITY, MIN_PRIORITY, NORM_PRIORITY`

Constructor Summary
`watchDog(CounterObject counter, finalScheduler scheduler)` The Constructor takes counterObject and finalScheduler as its parameters.The pointerTofinalScheduler points to the finalScheduler(Main thread) object and the pointerToCounterObject points to the counter object.CounterObject is passed so as to enable the watchdog to access the counter value and the finalScheduler is accessed to enable the watchdog to eset the main when it is not executing well.

Method Summary	
`void`	`run()` Run method calls the runWatchDog method when this thread is started.

`void`	`runWatchDog()` The runWatchDog method increaments the value of the counter.If the counter value reaches 1.2 seconds (i.e.One task might be in an infinite loop), then the main thread is killed and a new main thread is created.The tasks are re-loaded and then the thread is started by running the schedule method of the finalScheduler.

Methods inherited from class java.lang.Thread
`activeCount, checkAccess, countStackFrames, currentThread, destroy, dumpStack, enumerate, getContextClassLoader, getName, getPriority, getThreadGroup, interrupt, interrupted, isAlive, isDaemon, isInterrupted, join, join, join, resume, setContextClassLoader, setDaemon, setName, setPriority, sleep, sleep, start, stop, stop, suspend, toString, yield`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait`

Constructor Detail

watchDog

```
public watchDog(CounterObject counter,
                finalScheduler scheduler)
```

The Constructor takes counterObject and finalScheduler as its parameters.The pointerTofinalScheduler points to the finalScheduler(Main thread) object and the pointerToCounterObject points to the counter object.CounterObject is passed so as to enable the watchdog to access the counter value and the finalScheduler is accessed to enable the watchdog to eset the main when it is not executing well.

Method Detail

runWatchDog

```
public void runWatchDog()
```

The runWatchDog method increaments the value of the counter.If the counter value reaches 1.2 seconds (i.e.One task might be in an infinite loop), then the main thread is killed and a new main thread is created.The tasks are re-loaded and then the thread is started by running the schedule method of the finalScheduler.

---

run

`public void **run**()`

Run method calls the runWatchDog method when this thread is started.

Overrides:

run in class `java.lang.Thread`

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS — FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD — DETAIL: FIELD | CONSTR | METHOD

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class loadTasks

```
java.lang.Object
  |
  +--loadTasks
```

---

public class **loadTasks**
extends java.lang.Object

Used to load tasks to the linked list that is being scheduled.

Field Summary

`java.util.LinkedList`	`list` Construct a empty linked list that is used to store tasks

Constructor Summary

`loadTasks()`

Method Summary

`void`	`ClassLoad(java.lang.String className)` Method to load tasks to the linked list.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

list

```
public java.util.LinkedList list
```

Construct a empty linked list that is used to store tasks

Constructor Detail

loadTasks

```
public loadTasks()
```

Method Detail

ClassLoad

```
public void ClassLoad(java.lang.String className)
```

Method to load tasks to the linked list. All the tasks that will be added to the linked list with their schedule information. This list will be scheduled by the scheduler

---

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS **FRAMES** **NO FRAMES**
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class PrintWriter1

```
java.lang.Object
  |
  +--PrintWriter1
```

public class **PrintWriter1**
extends java.lang.Object

Class to read information from the text file. The file reader class is created to help read information from a text file. The text file is created to provide information about our tasks as to what tasks to schedule, how frequent are tasks to be scheduled and if the task is schedulable immediately or will be scheduled later. It is easier to use the text file because when we want to update the tasks, we only need to compile the task into a class file, and then update the text file, which does not need to be compiled.

Version:

1.0

Author:

Mpho Ntsimane

Field Summary

java.lang.String	**fileName** String representing the name of the file to be read.
java.lang.String	**line** String representing the line in the file to be read
java.lang.String	**methodName** String representing the method name of the file to be read.
java.io.BufferedReader	**readFromFile** Create a buffering character-input stream that uses a default-sized input buffer
java.lang.Boolean	**ScheduleFlag** Set when it is the time for the task to be scheduled
int	**scheduleTime** The frequency at which the task is supposed to be scheduled

Constructor Summary
`PrintWriter1()`

Method Summary	
void	`closeFile(java.lang.String readableFile)` Close the file after being read.Also takes a string readableFile as a parameter
void	`openFile(java.lang.String readableFile)` Method to open the file.Takes string readableFile(i.e.filename) as a parameter
void	`readingFile(java.lang.String readableFile)` Read the file and from the file get file name, method name, schedule frequency and schedule flag.This method also takes string readableFile as a parameter and executes using the while loop to check if there is still text in the file to read from

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

fileName

```
public java.lang.String fileName
```

String representing the name of the file to be read.

methodName

```
public java.lang.String methodName
```

String representing the method name of the file to be read.

line

```
public java.lang.String line
```

String representing the line in the file to be read

---

scheduleTime

```
public int scheduleTime
```

The frequency at which the task is supposed to be scheduled

ScheduleFlag

```
public java.lang.Boolean ScheduleFlag
```

Set when it is the time for the task to be scheduled

---

readFromFile

```
public java.io.BufferedReader readFromFile
```

Create a buffering character-input stream that uses a default-sized input buffer

Constructor Detail

PrintWriter1

```
public PrintWriter1()
```

Method Detail

openFile

```
public void openFile(java.lang.String readableFile)
```

Method to open the file.Takes string readableFile(i.e.filename) as a parameter

---

readingFile

`public void readingFile(java.lang.String readableFile)`

Read the file and from the file get file name, method name, schedule frequency and schedule flag.This method also takes string readableFile as a parameter and executes using the while loop to check if there is still text in the file to read from

---

closeFile

`public void closeFile(java.lang.String readableFile)`

Close the file after being read.Also takes a string readableFile as a parameter

---

Class addLinkedList1

```
java.lang.Object
  |
  +--addLinkedList1
```

public class **addLinkedList1**
extends java.lang.Object

Add linked list object adds elements to the linked lists created in the data object. This object's importance lies with the fact that when we want to add more data to the linked list we can just re-write this object and recompile it. Then this data will be added from this object to the data object. Every time we want to add new data to data object, we do not have to re-write the data object, rather we do it separately and just add to the always scheduled data object.

Constructor Summary

`addLinkedList1()`

Method Summary

`void`	`upDateLinkedList(dataObject myDataObject)` Method upDateLnikedList takes dataObject as a parameter.It first clears the linked list and add elements to the linked list.The add(int index, Object element) is then used to insert the specified element i.e.data, at the specified position in the linked list.Elements currently at that position (if any) and any subsequent elements will be shifted to the right i.e.adds one to their indices.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

addLinkedList1

```
public addLinkedList1()
```

Method Detail

upDateLinkedList

```
public void upDateLinkedList(dataObject myDataObject)
```

Method upDateLnikedList takes dataObject as a parameter.It first clears the linked list and add elements to the linked list.The add(int index, Object element) is then used to insert the specified element i.e.data, at the specified position in the linked list.Elements currently at that position (if any) and any subsequent elements will be shifted to the right i.e.adds one to their indices.

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class dataObject

```
java.lang.Object
  |
  +--dataObject
```

public class **dataObject**
extends java.lang.Object

Data object contains a pool of data used by the tasks. It can also be used to store that data created by tasks after being scheduled. The data used by the tasks are stored in the linked lists and data types is stored in their respective type linked list i.e. primitive double type is stored as Double object type by wrapping the primitive type by a Double object.

Constructor Summary
`dataObject()`

Method Summary	
void	`upDate()` Method update to copy data from new LL to the new LL.As tasks executes, they store data in the new linked list (i.e.Temporary storage) and when this method executes the new LL is stored as old linked list

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

dataObject

`public dataObject()`

Method Detail

upDate

`public void upDate()`

Method update to copy data from new LL to the new LL.As tasks executes, they store data in the new linked list (i.e.Temporary storage) and when this method executes the new LL is stored as old linked list

---

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS | FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

Class addNumbers

```
java.lang.Object
  |
  +--addNumbers
```

public class **addNumbers**
extends java.lang.Object

An example of a task file that will be scheduled.Task files takes dataObject as a parameter so that it is able to access and data and store data (if necessary).Because the linked list stores elements as Objects, data is accessed as object types (e.g.Double type) then unwrap it to primitive types (e.g.double type) then perform calculations and store it as object types.

Field Summary

int	i0 Interger primitive type
int	i1 Interger primitive type
int	i2 Interger primitive type
java.lang.Integer	newInt0 Integer Object type
java.lang.Integer	newInt1 Integer Object type

Constructor Summary

addNumbers()

Method Summary

`void`	`sum(dataObject myDataObject)` Method to perform mathematical operations, this method takes dataObject as a parameter and is used to access and store data from the data object

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Field Detail

newInt0

```
public java.lang.Integer newInt0
```

Integer Object type

---

newInt1

```
public java.lang.Integer newInt1
```

Integer Object type

---

i0

```
public int i0
```

Interger primitive type

---

i1

```
public int i1
```

Interger primitive type

---

i2

```
public int i2
```

Interger primitive type

Constructor Detail

addNumbers

```
public addNumbers()
```

Method Detail

sum

```
public void sum(dataObject myDataObject)
```

Method to perform mathematical operations, this method takes dataObject as a parameter and is used to access and store data from the data object

---

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class doubleClass

```
java.lang.Object
  |
  +--doubleClass
```

public class **doubleClass**
extends java.lang.Object

An example of a task that can be scheduled by our scheduler. It obtains data from the data object's linked list, uses them and stores new data on the linked list (Which can still be used by other tasks).

Field Summary

`java.lang.Double`	**`newDouble0`** Double Object type
`java.lang.Double`	**`newDouble1`** Double Object type

Constructor Summary

`doubleClass()`

Method Summary

void	`incDouble(dataObject myDataObject)` Method to make a simple addition of values obtained from the LL and stores the output in another index of the LL.Just like other tasks that uses dataObject, this method also passes dataobject as a parameter

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Field Detail

newDouble0

`public java.lang.Double **newDouble0**`

Double Object type

newDouble1

`public java.lang.Double **newDouble1**`

Double Object type

Constructor Detail

doubleClass

`public **doubleClass**()`

Method Detail

incDouble

`public void **incDouble**(dataObject myDataObject)`

Method to make a simple addition of values obtained from the LL and stores the output in another index of the LL.Just like other tasks that uses dataObject, this method also passes dataobject as a parameter

Class **Tree** **Deprecated** **Index** **Help**
PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS **FRAMES** **NO FRAMES**

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class getDouble2

```
java.lang.Object
  |
  +--getDouble2
```

public class **getDouble2**
extends java.lang.Object

An example of a task that can be scheduled by the scheduler.It obtains data from the data object's LL, uses them and stores new data on the linked list (Which can still be used by other tasks).

Field Summary

`double`	`d2` double primitive type
`java.lang.Double`	`newD2` Double Object type

Constructor Summary

`getDouble2()`

Method Summary

`void`	`getDouble(dataObject myDataObject)` Method to obtain a value from the linked list and print it out to the screen The value is obtained as an object type and is unwrapped to it primitive type before simple operations like add and subtract are performed on it.

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Field Detail

newD2

```
public java.lang.Double newD2
```

Double Object type

---

d2

```
public double d2
```

double primitive type

Constructor Detail

getDouble2

```
public getDouble2()
```

Method Detail

getDouble

```
public void getDouble(dataObject myDataObject)
```

Method to obtain a value from the linked list and print it out to the screen The value is obtained as an object type and is unwrapped to it primitive type before simple operations like add and subtract are performed on it.

Class getInt2

```
java.lang.Object
  |
  +--getInt2
```

public class **getInt2**
extends java.lang.Object

An example of a task that can be scheduled by our scheduler. It obtains data from the data object's linked list, uses them and stores new data on the linked list (Which can still be used by other tasks).

Field Summary

`int`	`i2` integer primitive type
`java.lang.Integer`	`newInt2` Integer object type

Constructor Summary

`getInt2()`

Method Summary

`void`	`getIt2(dataObject myDataObject)` Method to obtain a value from the linked list and print it out to the screen The value is obtained as an object type and is unwrapped to it primitive type before simple operations like add and subtract are performed on it.

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Field Detail

newInt2

`public java.lang.Integer newInt2`

Integer object type

---

i2

`public int i2`

integer primitive type

Constructor Detail

getInt2

`public getInt2()`

Method Detail

getIt2

`public void getIt2(dataObject myDataObject)`

Method to obtain a value from the linked list and print it out to the screen The value is obtained as an object type and is unwrapped to it primitive type before simple operations like add and subtract are performed on it.

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class linkedList

```
java.lang.Object
  |
  +--linkedList
```

public class **linkedList**
extends java.lang.Object

An object to create a linked list object. has one public field to create the linked list

Field Summary

`java.util.LinkedList`	**`list`** Linked list of unspecified length

Constructor Summary

`linkedList``()`

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

list

```
public java.util.LinkedList list
```

Linked list of unspecified length

Constructor Detail

linkedList

```
public linkedList()
```

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS | FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class MyOwnLoaderClass1

```
java.lang.Object
  |
  +--java.lang.ClassLoader
        |
        +--MyOwnLoaderClass1
```

---

public class **MyOwnLoaderClass1**
extends java.lang.ClassLoader

An object to create a user-defined ClassLoader Object

Constructor Summary

`MyOwnLoaderClass1()`

Methods inherited from class java.lang.ClassLoader

```
defineClass, defineClass, defineClass, definePackage, findClass,
findLibrary, findLoadedClass, findResource, findResources,
findSystemClass, getPackage, getPackages, getParent, getResource,
getResourceAsStream, getResources, getSystemClassLoader,
getSystemResource, getSystemResourceAsStream, getSystemResources,
loadClass, loadClass, resolveClass, setSigners
```

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Constructor Detail

MyOwnLoaderClass1

```
public MyOwnLoaderClass1()
```

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class CounterObject

```
java.lang.Object
  |
  +--CounterObject
```

public class **CounterObject**
extends java.lang.Object

This class creates an integer count with the initial value equals zero

Field Summary

`int`	`count` integer primitive type

Constructor Summary

`CounterObject()`

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll,
toString, wait, wait, wait
```

Field Detail

count

```
public int count
```

integer primitive type

Constructor Detail

CounterObject

```
public CounterObject()
```

Class **Tree** **Deprecated** **Index** **Help**

PREV CLASS NEXT CLASS FRAMES NO FRAMES
SUMMARY: INNER | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class classWithMethods

```
java.lang.Object
  |
  +--classWithMethods
```

public class **classWithMethods**
extends java.lang.Object

A class with undisclosed number of arrays for methods of the task and a pointer to the task. Also contained in this class is the schedule period/frequency and schedule flag of the task.

Constructor Summary

`classWithMethods()`

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

classWithMethods

```
public classWithMethods()
```

RunScheduler.java

```
//:RunScheduler.java
import java.awt.*;
import java.awt.event.*;
class schedule {
        CounterObject tempCounter = new CounterObject();
        finalScheduler scheduler = new finalScheduler(tempCounter);
        watchDog WatchDog = new watchDog(tempCounter,scheduler);
        /**Starts the watchdog and the main threads*/
        public schedule()
        {
                WatchDog.start();
                System.out.println("WatchDog thread started");

                scheduler.start();
                System.out.println("scheduler thread started");
        }
        /**Calls the two threads run methods*/
        public void runScheduler()
        {
                        scheduler.run();
                        WatchDog.run();
        }
}
/** Starts the main thread (i.e. finalScheduler.java) and
*the watchdog thread (i.e. watchDog.java)
* and Executes the two thread's run methods.
*@version: 1.0
*@Author: Mpho Ntsimane
*/
public class RunScheduler
{
        /**Creates a new instance of the schedule object. The schedule
        * object will start the two threads (finalScheduler and watchDog)
        * and executes their run methods.*/
        public static void main (String[] args) throws Exception{

                schedule schedule = new schedule();
                schedule.runScheduler();
        }
}
```

99 JavaCode

finalScheduler.java

```
//: finalScheduler.java
import java.lang.Class;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Array;
import java.lang.reflect.InvocationTargetException;
import java.util.Timer;
import java.util.TimerTask;
import java.util.Calendar;
import java.util.LinkedList;
import java.io.*;
import java.io.FileWriter;
import dataObject;
/**Scheduler program which uses a fileReader to read the names ofthe tasks to load and loadClass method of
* ClassLoader to load the tasks and the loaded task is scheduled at the specifiedfrequency.All the tasks that
* have been scheduled will be written in a text file "ScheduledTasks.TXT" specifying the tasks and the time
* it has been scheduled and the Errors will be logged in a text file "Error.txt" which also specifies the
* unscheduled tasks and the time they were supposed to be scheduled
* @version: 1.0
* @author: Mpho Ntsimane
*/

public class finalScheduler extends Thread
{
        /** Create an instance of PrintWriter1*/
        PrintWriter1 PrinterAndWriter = new PrintWriter1();
        /** Create an instance of loadTasks*/
        //loadTasks newTasks = new loadTasks();
        loadTasks1 loadTasks = new loadTasks1();
        /** Create an instance of LinkedList*/
        LinkedList linkedList1 = new LinkedList();
        //count counts = new count();
        /** Create an instance of classWithMethods*/
        classWithMethods TemporaryClass = new classWithMethods();
        /** Create an instance of classWithMethods*/
        classWithMethods TemporaryClass1 = new classWithMethods();
        /** Create an instance of MyOwnLoaderClass*/
        MyOwnLoaderClass1 Loader = new MyOwnLoaderClass1();
        /** Terminates the execution of a task when set true */

        public boolean TerminateFlag = false;
        /**The Boolean object corresponding to the primitive value false.*/
        public Boolean True = Boolean.valueOf("true");
```

```
/** The Boolean object corresponding to the primitive value false.*/
public Boolean False = Boolean.valueOf("false");
Calendar calendar;
CounterObject pointerToCounterObject;
/**integer value corresponding to the clock'second*/
public int counter;
/** Tasks are loaded to the linked list and the scheduler is scheduled and the counter value is
* determined from the counter object.The constructor takes CounterObject as an argument which
* parses the integer counter value to the scheduler.
*/
public finalScheduler(CounterObject counter)
{
        loadTasks.ClassLoad("TasksToLoad.txt");
        linkedList1 = loadTasks.list;
        Timer timer = new Timer(true);
        timer.scheduleAtFixedRate(new RemindedTasks(),0, 1000);
        pointerToCounterObject = counter;
}
/** Method to schedule loaded methods.This method takes newTasks as a parameter to enable it to
* schedule new tasks.As this method executes, it also resets the counter.If the counter is not
* reset, then the the watchdog thread will reset the counter after 1.2 seconds.When executing,
* this method will run through the linked list and check the schedule flag for every task.If
* set then the scheduler will give the task the processor.The scheduler will check the following
* conditions:-
*
* 1:    If the task is at index one of the linked list (i.e data object), then that object
*               takes null when invoked.
* 2:    If the flag set signals task at index two, then new tasks will be loaded to the scheduler
* 3:    All the other object will take data object as an argument when scheduled,
*
* Then their schedulable flag will be cleared after the task has been scheduled.
*/
public synchronized void scheduleMethod(loadTasks extraTasks)
{
        PrintWriter writeToFile;
        linkedList1 = loadTasks.list;
        int classIndex;
        try
        {
                pointerToCounterObject.count = 0;
                for (classIndex=0;classIndex<linkedList1.size();classIndex++)
                {
                        pointerToCounterObject.count = 0;
                        TemporaryClass = (classWithMethods)linkedList1.get(classIndex);
                        if (TemporaryClass.Schedulable == True)
                        {
```

```
                                        System.out.println( "List size is :"+ linkedList1.size());
                                        calendar = Calendar.getInstance();

                                        System.out.println(" Task is now Scheduled : " + calendar.getTime());

                                        Object SchedulableObject = TemporaryClass.pointerToInstance;

                                        System.out.println(" SchedulableObject :" + SchedulableObject);

                                        Object[] parameterArray = new Object[1];

                                        parameterArray[0] = (dataObject)(
                                                (classWithMethods)linkedList1.get(1)).pointerToInstance;

                                        System.out.println(
                                                " The parameters of the loaded class : " + parameterArray[0]);

                                        Object MethodOutput;
                                        if (classIndex==1)
                                        {
                                                MethodOutput = (String)TemporaryClass.MethodArray[0].invoke(
                                                        SchedulableObject,null);
                                        }
                                        else if (classIndex==2)
                                        {
                                                loadTasks.ClassLoad("ExtraTasks.txt");
                                                LinkedList extraList = loadTasks.list;
                                                for (int c=0;c<extraList.size();c++)
                                                {
                                                        TemporaryClass1 = loadTasks.TemporaryClass;
                                                        linkedList1.addLast(TemporaryClass1);
                                                }
                                        }
                                        else
                                        {
                                                MethodOutput = (String)TemporaryClass.MethodArray[0].invoke(
                                                        SchedulableObject,parameterArray);
                                        }
                                        /***Write To the File****/
                                        writeToFile= new PrintWriter(new FileWriter("ScheduledTasks.TXT",true));
                                        writeToFile.print(" Scheduled Class " + TemporaryClass.pointerToClass +
                                                "Scheduled Time " + calendar.getTime() + "\n");
                                        writeToFile.close();
                                        TemporaryClass.Schedulable = False;
                                }
                        }
```

```
			}
			/** setPriority Exception*/
			catch(SecurityException e2){
					System.out.println("SecurityException e2 Caught");
			}
			/** INVOKE EXCEPTIONS*/
			catch (IllegalAccessException e3){
					System.out.println("IllegalAccessException e3 Caught");
			}
			catch (IllegalArgumentException e4){
					System.out.println("IllegalArgumentException e4 Caught");
			}
			catch (InvocationTargetException e5){
					System.out.println("InvocationTargetException e5 Caught ");
			}
			/**Writing To A File Exceptions*/
			catch(IOException eIO){
					System.err.println(
							" Error in file ( Report File ) " + eIO.toString());
					System.exit(1);
			}
		}
		/** Run Method : When the terminate flag is not set, then this method
		* will schedule tasks by invoking schedule method*/
		public synchronized void run()
		{
			loadTasks tasks = new loadTasks();
			while (!TerminateFlag)
			{
					scheduleMethod(tasks);
			}
			System.exit(0);
			System.out.println("Thread Terminated...");
		}
		/**The remind task runs through the linked list and sets the schedule
		* flags true when the tasks are supposed to be scheduled
		*/
		class RemindedTasks extends TimerTask
		{
			Calendar calendar;
			int classIndex;
	public synchronized void run()
	{
		calendar = Calendar.getInstance();
					PrintWriter writeToFile;
					for (classIndex=0;classIndex<linkedList1.size();classIndex++)
```

```
                    {
                                TemporaryClass = (classWithMethods)linkedList1.get(classIndex);
                                if (TemporaryClass.SchedulePeriod == 0)
                                {
                                }
                                else if (counter%TemporaryClass.SchedulePeriod==0)
                                {
                                        if (TemporaryClass.Schedulable == True)
                                        {
                                                try
                                                {
                                                        writeToFile= new PrintWriter(
                                                                new FileWriter("Error.txt",true));
                                                        writeToFile.print(
                                                                " Error In Sceduling " +
                                                                TemporaryClass.pointerToClass +
                                                                "Supposed To Be Scheduled at : " +
                                                                calendar.getTime() + "\n");
                                                        writeToFile.close();
                                                        TemporaryClass.Schedulable = False;
                                                }
                                                catch(IOException eIO2)
                                                {
                                                        System.err.println(
                                                                "Error in file ( Error Log File ) " +
                                                                eIO2.toString());
                                                        System.exit(1);
                                                }
                                        }
                                        else
                                        {
                                                calendar = Calendar.getInstance();
                                                TemporaryClass.Schedulable = True;
                                        }
                                }
                    }
                    counter++;
  }
        }
}///:~
```

watchDog.java

```
//: watchDog.java
import java.util.Calendar;
import java.io.*;
import java.io.FileWriter;
/** Used to watch if the scheduler's loop is executed every second.If the loop is not executed after 1.2 second, then this class resets
* the scheduler.
* @version: 1.0
* @Author: Mpho Ntsimane
*/

public class watchDog extends Thread
{
        CounterObject pointerToCounterObject;
        finalScheduler pointerTofinalScheduler;
        newTasks task = new newTasks();
        PrintWriter1 PrinterAndWriter = new PrintWriter1();
        /**The Constructor takes counterObject and finalScheduler as its parameters. The pointerTofinalScheduler points to the
        * finalScheduler(Main thread) object and the pointerToCounterObject points to the counter object.CounterObject is
        * passed so as to enable the watchdog to access the counter value and the finalScheduler is accessed to enable the
        * watchdog to eset the main when it is not executing well.
        */
        public watchDog(CounterObject counter, finalScheduler scheduler)
        {
                 pointerTofinalScheduler = scheduler;
                pointerToCounterObject = counter;
        }

        /**The runWatchDog method increaments the value of the counter.If the counter value reaches 1.2 seconds (i.e.One
        * task might be in an infinite loop), then the main thread is killed and a new main thread is created.The tasks
        * are re-loaded and then the thread is started by running the schedule method of the finalScheduler.
        */
        public void runWatchDog()
        {
                pointerToCounterObject.count++;
                System.out.println(pointerToCounterObject.count);
                Calendar calendar;
                calendar = Calendar.getInstance();

                PrintWriter writeToFile;
                if (pointerToCounterObject.count>=1200)
                {
                        try
                        {
                                writeToFile= new PrintWriter(new FileWriter("Error.txt",true));

                                writeToFile.print(" Main Thread Killed at" + calendar.getTime() + "\n");
                                writeToFile.close();
                                pointerToCounterObject.count = 0;
                                CounterObject tempCounter = new CounterObject();
```

```
                                        finalScheduler newPointerTofinalScheduler = new finalScheduler(tempCounter);
                                        newPointerTofinalScheduler.start();
                                }
                                catch(IOException eIO2)
                                {
                                        System.err.println("Error in file ( Error Log File ) " + eIO2.toString());
                                        System.exit(1);
                                }
                        }
        }
        /** Run method calls the runWatchDog method when this thread is started.*/
        public void run()
        {
                for(;;)
                {
                        runWatchDog();
                }
        }
}
```

loadTasks.java

```
//: loadTasks.java
import java.util.LinkedList;
import java.util.Calendar;
import java.util.LinkedList;
import java.io.*;
import java.io.FileWriter;

/** Used to load tasks to the linked list that is being scheduled.
*@version: 1.0
*@Author: Mpho Ntsimane
*/

public class loadTasks1
{
        /** Contructor for the new instance of the PrintWriter1 class */
        PrintWriter1 PrinterAndWriter = new PrintWriter1();
        /** Contructor for the new instance of the Class With Methods Object */
        classWithMethods TemporaryClass;
        classWithMethods TemporaryClass1;
        classWithMethods TemporaryClass2;
        /** Contructor for the new instance of ClassLoader class */
        MyOwnLoaderClass1 Loader = new MyOwnLoaderClass1();
```

```
/** Construct a empty linked list that is used to store tasks */
public LinkedList list = new LinkedList();
linkedList linkedList;
public String ClassToLoad;
/** Method to load tasks to the linked list. All the tasks that will be added to the linked list with
* their schedule information. This list will be scheduled by the scheduler
*/
public void ClassLoad(String className)
{
        try
        {
                PrinterAndWriter.openFile(className);

                PrinterAndWriter.readingFile(className);

                while (PrinterAndWriter.line!=null)
                {
                        ClassToLoad =  PrinterAndWriter.fileName;
                        /**If the linked list is empty, add the task at index 1 of the Linked list
                        */
                        if (list.size() == 0)
                        {
                                TemporaryClass = new classWithMethods();
                                TemporaryClass.pointerToClass=Loader.loadClass(ClassToLoad);

                                TemporaryClass.MethodArray =
                                        TemporaryClass.pointerToClass.getDeclaredMethods();
                                TemporaryClass.pointerToInstance =
                                        TemporaryClass.pointerToClass.newInstance();
                                TemporaryClass.Schedulable=PrinterAndWriter.ScheduleFlag;

                                TemporaryClass.SchedulePeriod=PrinterAndWriter.scheduleTime;

                                TemporaryClass.fileName=PrinterAndWriter.fileName;
                                PrinterAndWriter.readingFile(className);
                                list.add(0,TemporaryClass);
                                System.out.println(" 1st element of the LL: " + TemporaryClass.fileName);
                        }
                        else
                        {
                                for (int i=0;i<list.size();)
                                {
                                        TemporaryClass1 = (classWithMethods)list.get(i);

                                        String file = TemporaryClass1.fileName;
                                        ClassToLoad =  PrinterAndWriter.fileName;
                                        /**If the task has been loaded before, remove it from the linked
                                        * list.
                                        */
                                        if (ClassToLoad.equalsIgnoreCase(file))
                                        {
                                                System.out.println(
                                                                "REMOVE THE EXISTING CLASS");
                                                list.remove(i);
                                                i++;
                                        }
                                        /** If it has never been loaded, continue scanning the whole
                                        * linked list.
                                        */
                                        else if(ClassToLoad != file)
                                        {
                                                i++;
                                        }

                                }
                                /**Then add the task at the last index of the linked list*/
                                TemporaryClass = new classWithMethods();

                                TemporaryClass.pointerToClass = Loader.loadClass(ClassToLoad);

                                TemporaryClass.MethodArray =
                                        TemporaryClass.pointerToClass.getDeclaredMethods();
                                TemporaryClass.pointerToInstance =
                                        TemporaryClass.pointerToClass.newInstance();
                                TemporaryClass.Schedulable = PrinterAndWriter.ScheduleFlag;

                                TemporaryClass.SchedulePeriod = PrinterAndWriter.scheduleTime;

                                TemporaryClass.fileName = PrinterAndWriter.fileName;

                                PrinterAndWriter.readingFile(className);
                                list.addLast(TemporaryClass);
                        }
                }
                System.out.println("======= List Size =======" + list.size());
                PrinterAndWriter.closeFile(className);
        }
        /**LOAD CLASS EXCEPTION(S)*/
        catch (ClassNotFoundException e1){
                System.out.println("ClassNotFoundException e1 Caught");
                }
                catch(SecurityException e11){
                        System.out.println("SecurityException e11 Caught");
                }
                /**InstantiationException*/
                catch(InstantiationException e12){
                        System.out.println("InstantiationException e12 Caught");
                }
                catch (IllegalAccessException e13){
                        System.out.println("IllegalAccessException e13 Caught");
                }
        }
}
```

PrintWirter1.java

```
//: PrintWriter1.java
import java.io.*;
import java.io.FileReader;
import java.util.*;
/** Class to read information from the text file.The file reader class is created to help read information from
* a text file. The text file is created to provide information about our tasks as to what tasks to schedule, how
* frequent are tasks to be scheduled and if the task is schedulable immediately or will be scheduled later. It
* is easier to use the text file because when we want to update the tasks, we only need to compile the task
* into a class file, and then update the text file, which does not need to be compiled.
* @version 1.0
* @author Mpho Ntsimane
*/

public class PrintWriter1
{
        /** String representing the name of the file to be read.*/
        public String fileName;
        /** String representing the method name of the file to be read.*/
        public String methodName;
        /** String representing the line in the file to be read */
        public String line;
        /** The frequency at which the task is supposed to be scheduled */
        public int scheduleTime;
        /** Set when it is the time for the task to be scheduled */
        public Boolean ScheduleFlag;
        /** Create a buffering character-input stream that uses a default-sized input buffer */
        public BufferedReader readFromFile;
        /** Method to open the file.Takes string readableFile(i.e.filename) as a parameter */
        public void openFile(String readableFile)
        {
```

```
		try
		{
			readFromFile = new BufferedReader(new FileReader(readableFile));
		}
		catch(FileNotFoundException e1)
		{
			System.out.println("FILE NOT FOUND e1");
		}
	}
	/**Read the file and from the file get file name, method name, schedule frequency and schedule flag.This
	* method also takes string readableFile as a parameter and executes using the while loop to check if there
	* is still text in the file to read from.
	*/
	public void readingFile(String readableFile)
	{
		try
		{
			line = readFromFile.readLine();
			if (line!=null)
			{
				StringTokenizer tokens = new StringTokenizer(line,":",false);
				fileName = tokens.nextToken();
				System.out.println(" Class To Be Loaded : " + fileName);
				methodName = tokens.nextToken();
				System.out.println(" Method To Be Scheduled : " + methodName);
				String priority =tokens.nextToken();
				scheduleTime = Integer.parseInt(priority);
				System.out.println(" SchedulingPriority is : " +scheduleTime);
				String schedule = tokens.nextToken();
				ScheduleFlag = Boolean.valueOf(schedule);
				System.out.println(" ScheduleFlag is : " +ScheduleFlag);
			}
		}
		catch(FileNotFoundException e1)
		{
			System.out.println("FILE NOT FOUND");
		}
		catch (IOException e)
		{
			System.err.println("Error in file" + e.toString());
			System.exit(1);
		}
	}
	/** Close the file after being read.Also takes a string readableFile as a parameter*/
	public void closeFile(String readableFile)
	{
			try
			{
				readFromFile.close();
			}
			catch (IOException e)
			{
				System.err.println("Error in file" + e.toString());
				System.exit(1);
			}
		}
}
```

AddLinkedList1.java

```
//: dataObject.java
import dataObject;
/**
*Add linked list object adds elements to the linked lists created in the data object. This object's importance lies with the fact that
*when we want to add more data to the linked list we can just re-write this object and recompile it. Then this data will be added from
*this object to the data object. Every time we want to add new data to data object, we do not have to re-write the data object, rather
*we do it separately and just add to the always scheduled data object.
* @version: 1.0
* @Author: Mpho Ntsimane
*/
public class addLinkedList1
{
		/** Method upDateLnikedList takes dataObject as a parameter.It first clears the linked list
		* and add elements to the linked list.The add(int index, Object element) is then used to
		* insert the specified element i.e.data, at the specified position in the linked list.
		* Elements currently at that position (if any) and any subsequent elements will be shifted
		* to the right i.e.adds one to their indices.
		*/
		public void upDateLinkedList(dataObject myDataObject)
		{
			myDataObject.newIntList.clear();
			myDataObject.newIntList.add(0,new Integer(1));
			myDataObject.newIntList.add(1,new Integer(1));
			myDataObject.newIntList.add(2,new Integer(1));

			myDataObject.oldIntList.clear();
			myDataObject.oldIntList.add(0,new Integer(1));
			myDataObject.oldIntList.add(1,new Integer(1));
			myDataObject.oldIntList.add(2,new Integer(1));

			myDataObject.newDoubleList.clear();
			myDataObject.newDoubleList.add(0,new Double(1.0));
			myDataObject.newDoubleList.add(1,new Double(1.0));
			myDataObject.newDoubleList.add(2,new Double(1.0));

			myDataObject.oldDoubleList.clear();
			myDataObject.oldDoubleList.add(0,new Double(1.0));
			myDataObject.oldDoubleList.add(1,new Double(1.0));
			myDataObject.oldDoubleList.add(2,new Double(1.0));
		}
}
```

dataObject.java

```
//: dataObject.java
import java.util.LinkedList;
/**Data object contains a pool of data used by the tasks. It can also be used to store that data created by tasks after being
*scheduled. The data used by the tasks are stored in the linked lists and data types is stored in their respective type linked list
*i.e. primitive double type is stored as Double object type by wrapping the primitive type by a Double object.
* @version: 1.0
* @Author: Mpho Ntsimane
*/
public class dataObject
{
		LinkedList newDoubleList = new LinkedList();
		LinkedList oldDoubleList = new LinkedList();

		LinkedList newIntList = new LinkedList();
		LinkedList oldIntList = new LinkedList();
		/**Method update to copy data from new LL to the new LL.As tasks executes,
		* they store data in the new linked list (i.e.Temporary storage) and when
		* this method executes the new LL is stored as old linked list
		*/
		public void upDate()
		{
			oldIntList.set(0, newIntList.get(0));
			oldIntList.set(1, newIntList.get(1));
			oldIntList.set(2, newIntList.get(2));

			oldDoubleList.set(0, newDoubleList.get(0));
			oldDoubleList.set(1, newDoubleList.get(1));
			oldDoubleList.set(2, newDoubleList.get(2));
		}
}
```

addNumbers.java

```
//:addNumbers.java
import dataObject;
```

```
/**An example of a task file that will be scheduled.Task files takes dataObject as a parameter so that
* it is able to access and data and store data (if necessary).Because the linked list stores elements
* as Objects, data is accessed as object types (e.g.Double type) then unwrap it to primitive types
* (e.g.double type) then perform calculations and store it as object types.
* @version: 1.0
* @Author: Mpho Ntsimane
*/
public class addNumbers
{
        /** Integer Object type */
        public Integer newInt0,newInt1;
        /** Interger primitive type */
        public int i0,i1,i2;
        /**Method to perform mathematical operations, this method takes
        * dataObject as a parameter and is used to access and store data from the data object
        */
        public void sum(dataObject myDataObject)
        {
                newInt0 = (Integer)myDataObject.oldIntList.get(0);
                i0 = newInt0.intValue();
                newInt1 = (Integer)myDataObject.oldIntList.get(1);
                i1 = newInt1.intValue();
                i2 = i0 + i1;
                myDataObject.oldIntList.set(2,new Integer(i2));
        }
}
```

doubleClass.java

```
//: doubleClass.java
import dataObject;
/** An example of a task that can be scheduled by our scheduler. It obtains data from the data object's linked list, uses them and
*stores new data on the linked list (Which can still be used by other tasks).
* @version: 1.0
* @Author: Mpho Ntsimane
*/

public class doubleClass
{
        /** Double Object type */
        public Double newDouble0,newDouble1;

        double d0,d1,d2;
        /**Method to make a simple addition of values obtained from the LL and stores the output in
        * another index of the LL.Just like other tasks that uses dataObject, this method also passes
        * dataobject as a parameter
        */
        public void incDouble(dataObject myDataObject){
                newDouble0 = (Double)myDataObject.oldDoubleList.get(0);
                d0 = newDouble0.doubleValue();
                newDouble1 = (Double)myDataObject.oldDoubleList.get(1);
                d1 = newDouble1.doubleValue();
                d2 = d0 + d1;
                myDataObject.oldDoubleList.set(2,new Double(d2));
        }
}
```

getDouble2.java

```
//: getDouble2.java
import dataObject;
/** An example of a task that can be scheduled by the scheduler.It obtains data from the data object's LL, uses them
* and stores new data on the linked list (Which can still be used by other tasks).
* @version: 1.0
* @Author: Mpho Ntsimane
*/
public class getDouble2
{
        /** Double Object type */
        public Double newD2;
        /** double primitive type */
        public double d2;
        /**Method to obtain a value from the linked list and print it out to the screen. The value is obtained as an object
        * type and is unwrapped to it primitive type before simple operations like add and subtract are performed on it.
        */
        public void getDouble(dataObject myDataObject)
        {
                newD2 = (Double)myDataObject.oldDoubleList.get(2);
                d2 = newD2.doubleValue();
                System.out.println(" The Stored Value From Index 3 of the list(DOUBLE) : " + d2);
        }
}
```

getInt2.java

```
//: getInt2.java
import dataObject;
/** An example of a task that can be scheduled by our scheduler. It obtains data from the data object's linked list, uses them
```

```
* and stores new data on the linked list (Which can still be used by other tasks).
* @version: 1.0
* @Author: Mpho Ntsimane
*/
public class getInt2
{
        /** Integer object type */
        public Integer newInt2;
        /** integer primitive type */
        public int i2;
        /**Method to obtain a value from the linked list and print it out to the screen. The value is obtained as an object type
        * and is unwrapped to it primitive type before simple operations like add and subtract are performed on it.
        */
        public void getIt2(dataObject myDataObject)
        {
                newInt2 = (Integer)myDataObject.oldIntList.get(2);
                i2 = newInt2.intValue();
        }
}
```

LinkedList.java

```
//: LinkedList.java
import java.util.LinkedList;
/** An object to create a linked list object. has one public field to create the linked list
*@version: 1.0
*@Author: Mpho Ntsimane
*/


public class linkedList
{
        /**Linked list of unspecified length*/
        public LinkedList list = new LinkedList();
}
```

MyOwnClassLoader.java

```
//: MyOwnLoaderClass1.java
import java.lang.*;
import java.lang.Class;
/** An object to create a user-defined ClassLoader Object
*@version: 1.0
*@Author: Mpho Ntsimane
*/
```

```
public class MyOwnLoaderClass1 extends ClassLoader{
}
```

CounterObject.java

```
//: CounterObject.java
/** This class creates an integer count with the initial value equals zero
*@version: 1.0
*@Author: Mpho Ntsimane
*/
public class CounterObject
{
        /**integer primitive type*/
        public int count;
}
```

ClassWithMethods.java

```
//: Schedule.java
import java.lang.reflect.Method;
/** A class with undisclosed number of arrays for methods of the task and a pointer to the task. Also contained
* in this class is the schedule period/frequency and schedule flag of the task.
*@version: 1.0
*@Author: Mpho Ntsimane
*/

public class classWithMethods
{
        /** A pointer to the task object instance*/
        Class pointerToClass;
        String fileName;
        /**An array to load the methods of the task loaded. Its size is determined by the number of methods the
        * loaded task has
        */
        Method[] MethodArray;
        /** The pointer to the task object*/
        Object pointerToInstance;
        /** Schedule frequency/period : determines the frequency at whic the task will be scheduled */
        int SchedulePeriod;
        /** Schedulable Flag : is set when the task is ready to be scheduled */
        Boolean Schedulable;
}
```