A Design Pattern for Using Non-developmental Items in Real-Time Java

T. W. Otani, M. Auguston, T. S. Cook,
D. Drusinsky, J. B. Michael and M. Shing
Computer Science Department
Naval Postgraduate School
Monterey, California 93943, USA
(831) 656-3391
{twotani, maugusto, tscoo1, ddrusins, bmichael, shing}@nps.edu

ABSTRACT
This paper addresses the need to reduce the difficulties in developing time-constrained Java applications. We present a design pattern for a class of time-constrained real-time applications that allows developers to use (and re-use) Java code libraries and non-developmental items (NDI). The proposed design pattern simplifies the implementation of the time-constrained tasks substantially by not requiring the use of no-heap real-time threads. We tested the design pattern with the Java Real-Time System (RTS) 2.0 from the Sun Microsystems. This paper also presents a simple methodology for determining the appropriate values for the RTS run-time parameters (thread priorities, memory usage, process load, and task deadlines) in order to ensure the deterministic execution of the time-constrained tasks.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features - *concurrent programming structures, patterns, control structures.*

General Terms
Measurement, Performance, Design, Experimentation, Languages.

Keywords
Real-time system, Java programming language, Garbage collection, Design pattern, Non-developmental items.

1. INTRODUCTION
In response to the increasing popularity of the Java programming language for time-constrained applications, the Real-Time for Java Expert Group (RTJEG) developed the *Real-Time Specification for Java* (RTSJ), an extension of the *Java Language Specification* and the *Java Virtual Machine Specification*, to enable "the creation, verification, analysis, execution, and management of Java threads whose correctness conditions include timeliness constraints (also known as real-time threads)" [1]. Having identified the garbage collected heaps as the major source of unpredictable latencies in Java applications, the RTJEG introduced two new features in the Java memory model (immortal memory and scoped memory) as well as a new feature in the Java thread model (no-heap real-time thread) to allow Java programs to allocate objects outside the garbage collected heap and to permit real-time threads to run without interference from the garbage collector. These extensions, however, have resulted in a complex programming model that is difficult to understand and hard to analyze [2-6]. Moreover, the extensions take away two of the most valuable assets of the Java programming language: the abundance of free, open source, and commercial code libraries and components, and the large number of skilled programmers in the Java development community.

This paper addresses the need to reduce the difficulties in developing time-constrained Java applications using the no-heap real-time thread and scoped memory features provided by the Real-Time Java Extension. It focuses on a class of real-time applications whose computations must be terminated by their hard deadlines and have to return the best approximations to their clients if they cannot finish their computations by the deadlines. Many of these computations are iterative in nature, resulting in successive approximations that converge to the exact solution only in the limit. Examples include Newton's method, the bisection method, and the Jacobi iteration for solving large complex system of ordinary and differential equations, inexact computations for large-scale optimizations, as well as complex search methods for pattern matching and discrete optimizations. The computer programs for implementing these algorithms are usually written by scientists or operations researchers who have only limited understanding of real-time systems and lack the skill to produce correct Java programs using the immortal/scoped memories and no-heap real-time threads.

We present a design pattern for the aforementioned class of real-time applications. This design pattern enables developers to use (and re-use) Java code libraries and components in the time-constrained applications without employing no-heap real-time threads. We tested the proposed design pattern with the Java Real-Time System (RTS) 2.0 from the Sun Microsystems [7, 8].

JTRES '07, September 26-88, 2007 Vienna, Austria

The rest of the paper is organized as follows. Section 2 discusses the drawbacks of the no-heap real-time thread solution and presents a design pattern that uses heaps. Section 3 describes a methodology for determining the appropriate values for the RTS run-time parameters (thread priorities, memory usage, process load, and task deadlines) in order to ensure the deterministic execution of the time-constrained tasks, Section 4 discusses related work and Section 5 draws some conclusions.

2. DESIGN PATTERN FOR REAL-TIME THREAD WITH REAL-TIME GARGAGE COLLECTOR

In this section, we first discuss the difficulties involved in using no-heap real-time threads for the class of applications we identified in Section 1. Then we present an overview of our proposed design pattern and experimental results. (A detailed description of the pattern is available in the Appendix.)

2.1 Drawbacks of the no-heap real-time thread solution

When we require absolutely no interference from the garbage collector, we can use no-heap real-time threads. Such an approach would work smoothly if the upper bound of the memory usage is known a priori. If the amount of required memory is small, then we can allocate it in the call stack. Otherwise, we can allocate it in the scoped or immortal memory. But for many types of applications that manipulate data structures, such an a priori upper bound is not known. Adopting no-heap real-time threads for such data-intensive applications calls for the use of wait-free read and write queues to pass the data back and forth between the heap and the no-heap real-time threads.

Consider a hypothetical application that keeps track of very fast moving objects in the sky. We have an external sensor that detects a flying object, and the tracking data for the detected object is stored in a data structure (inside the heap memory). For each detected object, we want to discriminate whether it is a foe or a friend. There is a hard real-time requirement for the discriminator so that the system will be able to intercept the foes in time. To meet the hard real-time deadline, we can run the discriminators as no-heap real-time threads so that there will be no interference from the garbage collector. To pass an object's tracking data to the discriminator, we must use a WaitFreeReadQueue. And if the discriminator modifies the tracking data and we want to put the updated information back into the data structure, we need to use a WaitFreeWriteQueue. Notice that the data passed to the discriminator is a clone of the actual data because no-heap real-time threads must reside in the immortal memory and cannot access any data in the heap directly. Figure 1 illustrates the software architecture.

Using such wait-free read and write queues correctly and managing the complexity of allocating objects (threads) in the immortal/scoped memory is well beyond the expected skill level of the majority of Java programmers. It begets the questions: *Isn't there a simple design pattern that is easy to follow for the majority of Java programmers? A design pattern that won't impose a steep learning curve? A design pattern for the masses?* Our answer is yes.

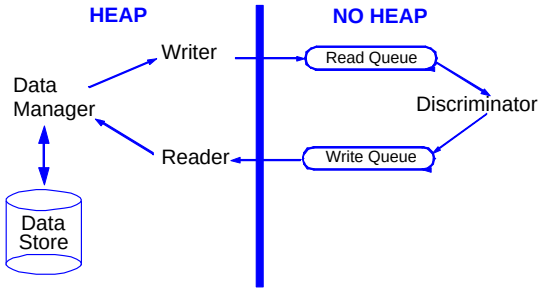

Figure 1. Software architecture for passing data back and forth between the heap and no-heap real-time threads.

2.2 The Sun Java Real-Time System

We propose a simple design pattern for a real-time Java system that does not require the use of no-heap real-time threads, and test the pattern with a simple prototype that runs on the Sun RTS. The prototype utilizes the following two parameters of the Sun Java Real-Time System garbage collector [9].

2.2.1 RTGCCriticalPriority

The RTGCCriticalPriority runtime parameter is most significant in the Sun Java RTS V2 release for ensuring the determinism of time-critical threads. A thread with the assigned priority higher than RTGCCriticalPriority is called the *critical real-time thread*. RTGC starts running at RTGCNormalPriority (whose default value is the minimum priority for the real-time threads). The auto-tuning mechanism attempts to start RTGC soon enough so that the garbage collection completes before reaching the memory threshold (RTGCCriticalReservedBytes), which will result in bumping up the priority of RTGC to RTGCCriticalPriority.

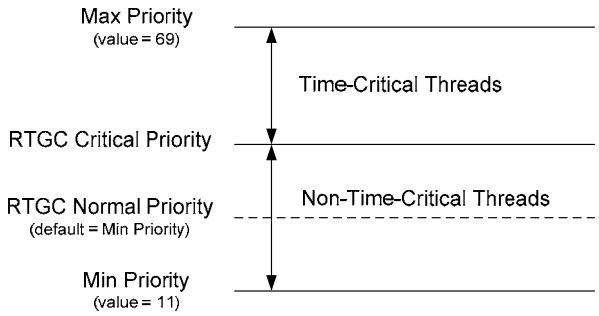

Figure 2. Java real-time thread classification

2.2.2 RTGCCriticalReservedBytes

To aid the RTGC in ensuring the deterministic behavior of all the time-critical threads, the programmer needs to specify the second runtime parameter RTGCCriticalReservedBytes (the default value is 0). When the free memory becomes less than the value set for the RTGCCriticalReservedBytes, RTGC runs at the RTGCCriticalPriority, using all CPU cycles not used by the time-critical threads. This prevents all other threads (non-time-critical real-time threads and non-real-time threads) from allocating CPU cycles and memory, and caused them to be blocked. It is important to be aware that critical threads with a higher priority

can still get blocked by the lower priority RTGC if there is not enough memory for the critical threads to run. In general, we want to set the RTGCCriticalReservedBytes just high enough to ensure that the critical threads do not get preempted by the RTGC due to the lack of free memory. If RTGCCriticalReservedBytes is set too high, the RTGC will run more frequently, thereby preventing the lower priority threads from running. This will reduce the overall throughput. The important points to remember regarding the value for RTGCCriticalReservedBytes are as follows:

- RTGCCriticalReservedBytes too high ⇒ lower throughput
- RTGCCriticalReservedBytes too low ⇒ determinism compromised

2.3 The Shadow Pattern

The proposed design pattern is defined by the following two key features:

- Real-time threads are divided into two groups, with the threads in the first group having a priority higher than the one assigned to the RTGC and the threads in the second group having a priority lower than the one assigned to the RTGC.
- In case the set deadline is missed, a predetermined or approximate value (e.g., via table-lookup or most recent value of the approximation) is used as the result of the computation.

Figure 3 shows the collaboration diagram of the design pattern. Only the Shadow threads run in a priority lower than the RTGC priority.

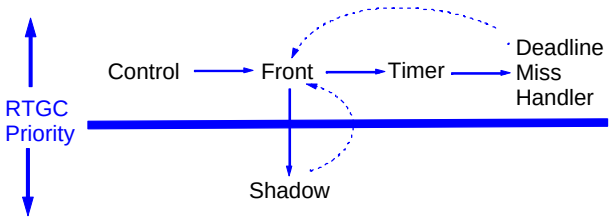

Figure 3. Collaboration diagram of the Shadow pattern.

We name our design pattern *Shadow* because a real-time thread in the second group act as a shadow for the corresponding real-time thread in the first group and performs the actual computation. The shadow threads do not interact directly with any other objects in the application and can be used to encapsulate reusable components. Because the threads in the first group are the ones that interact with the other objects in the application, they are called the *fronts*.

The Control is the main controller of the program, and it manages data objects (e.g., data of the flying objects). For each task (e.g., determining the flying object is a foe or a friend) associated to a data item, the Control creates a front thread and assigns the task to it. The front in turn delegates the requested task to its shadow and sets the deadline. The shadow carries out the task and reports the result back to the front.

We run the front and shadow threads at a priority higher and lower, respectively, than the one for the RTGC. We call the front threads *critical threads* and the shadow threads *non-critical threads*.

The defining feature of the shadow design pattern is the requirement that only the non-critical shadow threads consume uncertain amount of memory in the heap. The critical front threads consume heap memory with known upper bounds on the maximum heap usage and the heap mutation rate (e.g., only performing simple table-lookup or keeping track of intermediate results using a fixed number of data objects). This requirement leads to an architecture that ensures the critical threads will not get preempted by the RTGC, thus guaranteeing the determinism of the critical threads.

The shadow threads carry out the computational tasks on behalf of the front threads. A result is reported back to the front thread if the task is completed before the deadline. Alternatively, the shadow thread may choose to report the intermediate results to the front thread periodically as it continues to work towards the final result. If the deadline is missed, then the front thread kills its shadow and uses a predetermined value via table-lookup or the best value reported by the shadow thread so far as its final result.

A shadow thread can miss the deadline in two ways. First, as the shadow thread runs in a lower priority, it can be preempted and paused to wait for the RTGC to complete its garbage collection. The shadow misses the deadline when the pause becomes too long. Second, when the deadline is set too soon, the shadow thread may simply not have enough time to complete the assigned task even without any interruptions from the RTGC.

We use a timer to keep track of the deadline. The OneShotTimer class from the standard javax.realtime package is appropriate for the timer. The deadline can be set by designating the time duration, an instance of the standard RelativeTime class. When the deadline is missed, the timer transfers the control asynchronously to a deadline miss handler. The deadline miss handler then notifies the front thread that the deadline is missed. At that point, the front thread returns the preset value as the result of computation. When the actual result is computed by the shadow thread before the deadline, the actual result is used by the front thread.

2.4 Experimental Results

We ran a small test program using the Sun Java Real-Time System V2 release on a Sun Blade™ 2500 workstation (with a 1.6-GHz UltraSPARC IIIi processors with 1 MB of Level 2 cache, and 2GB RAM) to study our proposed design and to confirm its viability. In this section, we describe the test program and report the results of running the test program under different parameters such as the number of fronts, pause time between the creation of fronts, and the deadline. In Section 3, we describe the methodology for determining appropriate values for these parameters.

The Control is implemented as a RealtimeThread and its run method is defined as follows:

```
public void run( ) {
  for (int i = 0; i < N; i++) {
    DataItem node = new DataItem(i);
    front[i] = new Front(this, i, node);
    frontCnt++;
  }
```

```
    for (int i = 0; i < N) {
      front[i].start();
      /* Point A - Place delay here */
    }
  }
```

We are using an array to keep track of the front threads. Every index position of this array is a non-null value as it points to an instance of the Front class. When a front finishes its computation, it calls the Control's *workDone( )* method to report the completion of the assigned task. This will result in setting the corresponding index position to null, thereby turning the used heap memory into garbage.

At Point A in the code, we can place a time delay after a front is started. Placing no delay means the program will run all front threads simultaneously. This could lead to an OutOfMemory exception when N, the total number of fronts, becomes larger than a certain threshold. The reason is that the priority of Control is higher than the one for RTGC. As Control creates and starts more and more fronts, more and more memory gets consumed but there is no garbage to collect because there is no index position in the array that is set to null. In other words, the front threads never have a chance to call the Control's *workDone( )* method.

If we insert some delay at Point A in the code, then it becomes possible for the fronts to call the Control's *workDone( )* method to turn themselves and memory allocated by the corresponding shadows into garbage for the RTGC to collect.

A front thread performs the discrimination operation on a given data item. The actual work of discrimination is done by its associated shadow thread. In this test program, we simulate the computation by calling a method of the DataItem object. This stub method will go through a "dummy" computation loop. When the computation is complete, it calls its controlling front's *reportFinal( )* method to report the full result, which will, in turn, cause the front to invoke the Control's *workDone( )* method. Alternatively, the shadow thread may call the front's *reportProgress( )* method periodically to report the intermediate results to the front thread as it continues to work towards the final result.

The deadline is set by designating the time duration (RelativeTime that specifies the time duration such as 2 ms) using a OneShotTimer. When the time is up, its associated asynchronous event handler DeadlineMissHandler calls the front's *reportNominal( )* method to report the nominal result, which will, in turn, cause the front to invoke the Control's *workDone( )* method.

The front can get the result in two ways. The first is the full result, that is, the actual computation result received from its shadow via the *reportFinal( )*. In this case, the OneShotTimer object is killed. The second is the nominal result. This result is used when the timeout occurs. In this case, the associated shadow is killed.

2.4.1 Test 1

The first set of tests is run by placing no delays (delay time = 0). The test results are shown in Table 1. As expected, the table shows that we get an OutOfMemory exception when N = 1500. Placing no delays in between successive starting of fronts also means that none of the low-priority shadows can run until all N fronts have been started, resulting in the large number of missed deadlines. As we increase the deadline, the number of timeouts (i.e. missed deadlines by the shadow thread) decreases. When we increase the deadline the shadows have more time to complete their computations. This will result in having fewer timeouts for the same number of fronts. For example, with 200 fronts, we see anywhere from 35 to 200 occurrences of timeouts when the deadline is set to 100 ms. When the deadline is increased to 500 ms, we see no timeouts at all.

Table 1: No delay between the starting of front threads

Deadline (ms)	N (# of front threads)	Results (# of timeouts)
20	100	79 ~ 100
	200	200
	500	500
	1000	1000
	1500	OutOfMemory
50	100	28 ~ 96
	200	142 ~ 200
	500	500
	1000	1000
	1500	OutOfMemory
100	100	0 ~ 60
	200	35 ~ 200
	500	500
	1000	1000
	1500	OutOfMemory
500	100	0
	200	0
	500	184 ~ 434
	1000	998 ~ 1000
	1500	OutOfMemory

2.4.2 Test 2

In the second set of tests, we place a delay of 5 ms at Point A in the code. By placing a delay, we expect to have fewer timeouts, and to be able to run a larger number of fronts without getting an OutOfMemory exception because the RTGC will be able to reclaim garbage. With no delay, the run method of Control never gets interrupted, and there will be no null pointers in the front array. With a delay, the run method can get interrupted and the fronts get a chance to call the Control's *workDone( )* method, which will reset the content of the front array, at the index position that corresponds to the calling front, to null. This will result in the RTGC reclaiming memory allocated by the corresponding shadow. As heap memory spaces are recycled, we can avoid the OutOfMemory exceptions we have seen in the first set of tests.

Table 2 shows the results of test runs with the delay of 5 ms. There is zero timeout when the number of fronts (N) is less than or equal to 1000. When N is 1500, we still get an OutOfMemory exception regardless of the values for the deadline. This means that the 5 ms delay time is simply not large enough to slow down the starting rate of the front threads so that enough fronts can call the Control's *workDone( )* method to turn themselves into garbage for the RTGC to collect.

Table 2: Delay of 5 ms between the starting of front threads

Deadline (ms)	N (# of front threads)	Results (# of timeouts)
20	100	0
	200	0
	500	0
	1000	0
	1500	OutOfMemory
50	100	0
	200	0
	500	0
	1000	0
	1500	OutOfMemory
500	100	0
	200	0
	500	0
	1000	0
	1500	OutOfMemory

2.4.3 Test 3

Test 3 is the same as Test 2 but with the delay time set to 50 ms. This increase in the pause time enables the RTGC to perform garbage collection, thereby resulting in the elimination of the OutOfMemory exception. In this test run, when N = 1500, the garbage collection occurred once, and there are only 5 or less timeouts when deadline = 20 ms and N = 1500. Table 3 shows the results.

Table 3: Delay of 50 ms between the starting of front threads

Deadline (ms)	N (# of front threads)	Results (# of timeouts)
20	100	0
	200	0
	500	0
	1000	0
	1500	1 ~ 5 (1 GC)
50	100	0
	200	0
	500	0
	1000	0
	1500	0 (1 GC)
500	100	0
	200	0
	500	0
	1000	0
	1500	0 (1 GC)

3. METHODOLOGY FOR DETERMINING RUN-TIME PARAMETERS

The goal, when implementing our proposed design for the actual program, is to set the necessary parameters so that the number of timeouts (T) is minimized: fewer timeouts means better quality of the final results. In an ideal situation, T should be equal to 0, that is, no timeouts occur. We observe that three parameters are important for determining the frequency of timeouts. They are the pause time (P) between the starting of the fronts, the deadline (D) for the shadows to complete the designated task, and the memory usage (M) of the shadows.

If we set P = 0, then the value of M determines the total number of fronts (N) (and their corresponding shadows and timeout handlers) that can be executed concurrently without causing an OutOfMemory exception. By increasing the value for D, we can decrease the number of timeouts to reach the point where T would be 0. Table 1, for example, shows that by setting D = 500, we can run 200 fronts without any occurrence of timeouts, and Table 3 shows that we can run 1500 fronts without any timeouts by setting P to 50 or longer.

In a typical real-time application, the upper bound for D is given as a system requirement; that is, we want a guaranteed performance of completing a critical task in no more than D time units. If D is given, we can attain T = 0 by determining the appropriate values for P and M. If the upper bound for M is known, then we can increase the value of P until T becomes 0, which, in turn, limits the number of concurrent critical threads the application can run simultaneously. If the upper bound for P is known, then we can determine the maximum value for M while maintaining T = 0. This, in turn, may prevent the use of certain algorithms or code libraries.

Table 4 summarizes the concepts. The "You Can/Need To" column specifies what the system designer needs to or can do in order to achieve no timeouts (T = 0) for the given parameters listed in the Given column.

TABLE 4: The "You Can/Need To" column specifies what you need to or can do in order to achieve no timeouts (T = 0) for the given parameters listed in the Given column

To Achieve T = 0	
Given the values for	**You Can/Need To**
P and D	Determine the maximum value for M. If a discriminator requires less than this value, then we can achieve T = 0. If a discriminator requires more, then we need to adjust the values of P or D to get T = 0.
P and M	Determine the threshold value for D. If this value is not acceptable, then we need to increase the value for P.
D and M	Determine the threshold value for P. Any value below this threshold will increase the occurrences of timeouts.
T - timeouts; P - pause time (inter-arrival time); M - memory requirements per task; D - deadline	

4. RELATED WORK

In [10], Liu *et. al.* introduced the notion of imprecise computation, where each periodic task is logically decomposed into two parts: a mandatory part that must be completed by the deadline to produce an acceptable result, followed by an optional part to reduce the error of the result produced by the mandatory part if the schedule permits. The focus of [10] is to produce a feasible schedule that guarantees the timely completion of the mandatory part of every periodic task while minimizing the overall error (i.e. the average error or the total error) of the tasks. Our work is concerned with the timely reporting of acceptable results to the client. The refined results obtained from additional computation beyond the specified deadline has no value to the client. However, in the case that the application requires the shadow threads to periodically report the intermediate results to the front threads as they continue to work towards the final results, one may apply the imprecise computation scheduling to improve the fairness and timely progress among the shadow threads.

Sha *et. al.* introduced an architectural framework, called the Simplex Architecture, to support the use of commercial off-the-shelf (COTS) software in high-reliability systems [11-13]. Under the Simplex Architecture, a system is partitioned into a high-assurance portion and a high-performance portion. The high-assurance kernel monitors the system state and takes over the computation using the software module in the high-assurance portion of the system if it detects any fault (e.g. missed deadline) in the high-performance portion of the system. Both the Simplex Architecture and the shadow pattern work in the same spirit of the safety executive pattern, relying on the run-time monitoring of constraint violation and the use of analytic redundancy of software to tolerate faults. Sha *et al.*'s work focused on the high-level structure for a fault tolerant architecture and did not provide implementation details of their framework in their publications. In contrast, we introduce a practical pattern for creating Java real-time applications by the masses.

5. CONCLUSION

This paper addressed the need to simplify the implementation for a class of time-constrained applications. We proposed a design pattern in which the developers can use (and re-use) Java code libraries and components developed for non-time-constrained applications in implementing time-constrained applications. In our proposed design pattern, computational tasks either return the final results by the hard deadlines or the best approximations of them if the final results cannot be computed by the deadlines. The key requirements for using the proposed design pattern include:

(1) The availability of the real-time garbage collector (RTGC) with either an assignable priority lower than the critical thread of the application or a deterministic garbage collection behavior.

(2) A known upper bound on the maximum heap usage and the heap mutation rate of the critical thread. This is easily obtainable because the critical front threads (and the associated timers and deadline miss handlers) only perform simple table-lookup or keeping track of intermediate results using a fixed number of data objects.

Our prototype with the Sun RTS uses a RTGC with an assignable priority. This allows a uniform treatment for the priorities of all real-time threads (including the RTGC) and simplifies the schedulability analysis of the real-time software.

Our experiment showed that the proposed pattern can be used to implement predictable time-constrained applications with deadlines greater than or equal to 20 ms.

To improve the quality of the results returned by the time-constrained application, this paper also described the three run-time parameters that will affect the timeliness of the low-priority shadow threads and presented a methodology to determine, empirically, the tradeoff between the pause time (P), deadline (D) and the memory usage of the shadow threads (M) to minimize the number of timeouts experienced by the application.

There are two possible areas for future efforts. The first is the testing for the applicability of our design pattern to different types of garbage collectors. For instance, IBM WebSphere supports the Metronome garbage collector whose priority is not assignable by the programmer. We believe the proposed Shadow design pattern would work well with the Metronome garbage collector. We ran some preliminary experiments with favorable results but need to perform more tests.

The second is the undertaking of more elaborate testing methodologies. For the tests we reported in this paper, we simply put a fixed amount of pause time between the creations of the front threads. We would like to carry out the same tests by using more complex stochastic models.

6. ACKNOWLEDGMENTS

The authors thank Greg Bollella for the insightful discussions about the Sun Java Real-Time System. The research reported in this article was funded in part by a grant from the U.S. Missile Defense Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotations thereon.

7. REFERENCES

[1] Bollella, G., et al. *The Real-Time Specification for Java*, Addison-Wesley, 2001. Available at http://www.rtsj.org/specjavadoc/book_index.html

[2] Bollella, G., et al. Programming with non-heap memory in the real time specification for Java. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications* (Anaheim, CA, USA, 2003). ACM Press, New York, NY, 2003, 361 – 369.

[3] Kwon, J., Wellings, A., and King, S. Ravenscar-Java: a high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande JGI '02* (Seattle, Washington, USA, November 3-5, 2002). ACM Press, New York, NY, 2002, 131-140.

[4] Laukkanen, M. *Real-time Java—Memory Management.* Seminar on Real-time Java, Department of Computer Science, University of Helsinki, April 2001.

[5] Pizlo, F., Fox, J., Holmes, D. Vitek, J. Real-time Java scoped memory: design patterns, semantics. In *Proceedings of the IEEE International Symposium on Object-oriented Real-Time Distributed Computing (ISORC'04)* (Vienna, Austria, May 2004), 101-112.

[6] Potanin, A., Noble, J., Zhao, T., and Vitek, J. A High Integrity Profile for Memory Safe Programming in Real-time Java. In *Proceedings of the 3rd workshop on Java Technologies for Real-time and Embedded Systems* (San Diego, CA, USA, October 2005).

[7] Cook, T.S., Drusinsky, D., Michael, J.B., Otani, T.W., and Shing, M. *Design of Preliminary Experiments with the Sun Java Real-Time System*, Technical Report NPS-CS-06-010, Naval Postgraduate School, Monterey, CA, 2006.

[8] Auguston, M., Cook, T.S., Drusinsky, D., Michael, J.B., Otani, T.W., and Shing, M., *Experiments with Sun Java Real-Time System - Part II*, Technical Report NPS-CS-07-005, Naval Postgraduate School, Monterey, CA, 2007.

[9] *Sun Java Real-Time System 2.0 Garbage Collection*, Sun Microsystems Inc., Dec. 6, 2006

[10] Liu, J.W.S., Shih, W.-K., Lin, K.-J., Bettati, R., and Chung, J.-Y. Imprecise computations. In *Proceedings of the IEEE, 82*, 1(Jan. 1994), 83-94.

[11] Gagliardi, M., Rajkumar, R., ,Sha, L. Designing for evolvability: building blocks for evolvable real-time systems. In *Proc. 1996 IEEE Real-Time Technology and Applications Symposium* (10-12 June 1996), 100-109

[12] Sha, L., Goodenough, J. B., and Pollak, B. Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems. In *Crosstalk The Journal of Defense Software Engineering* (April, 1998), 7 – 10.

[13] Seto, D., Krogh, B., Sha, L., Chutinan, A. The Simplex architecture for safe online control system upgrades. In Proc. 1998 American Control Conference, Vol 6 (24-26 June 1998), 3504-3508

8. APPENDIX

In this appendix, we present our proposed pattern in the typical design pattern documentation format for easy reference and use.

Name: Shadow

Classification: Behavioral

Intent:

Ensure a time-constrained task returns an acceptable result to the client by the task deadline.

Motivation:

Consider for example an airspace defense system that keeps track of very fast moving objects in the sky. The system uses external sensors to detect flying objects and stores the tracking data for the detected objects in a data structure (inside the heap memory). For each detected object, the system calls a discriminator to determine whether it is a foe or a friend. There are many discriminators for the system to choose from, ranging from simple table-lookup that is fast but not very accurate, to feature-based analysis that is very accurate but very time- and memory-consuming.

There is a hard real-time requirement for the discriminator so that the system will be able to intercept the foes in time. To meet the hard real-time deadline, the system can run the discriminators as no-heap real-time threads so that there will be no interference from the garbage collector. Such a design will require the copying of tracking data between the heap memory and the immortal/scoped memory, and the passing of the cloned data to and from the no-heap real-time thread via wait-free read and write queues.

Using of wait-free read and write queues correctly and managing the complexity of allocating objects (threads) in the immortal/scoped memory is well beyond the expected skill level of the majority of Java programmers. Moreover, it will prevent the direct use of non-developmental items that are originally written for non-real-time Java applications.

The shadow pattern addresses the need to reduce the difficulties in developing time-constrained Java applications using the no-heap real-time thread and scoped memory features provided by the Real-Time Java Extension.

Applicability:

Use the pattern for real-time application whose computations must be terminated by their hard deadlines, and have to return the best approximations to their clients if they cannot finish their computations by the deadlines.

The approximate solution must be obtainable in time strictly less than the deadline and in heap memory with known upper bounds on the maximum heap usage and the heap mutation rate (e.g., only performing simple table-lookup or keeping track of intermediate results using a fixed number of data objects).

The real-time application must run in a real-time Java system with a real-time garbage collector (RTGC) that has either an assignable priority lower than the critical thread of the application or a deterministic garbage collection behavior.

Structure:

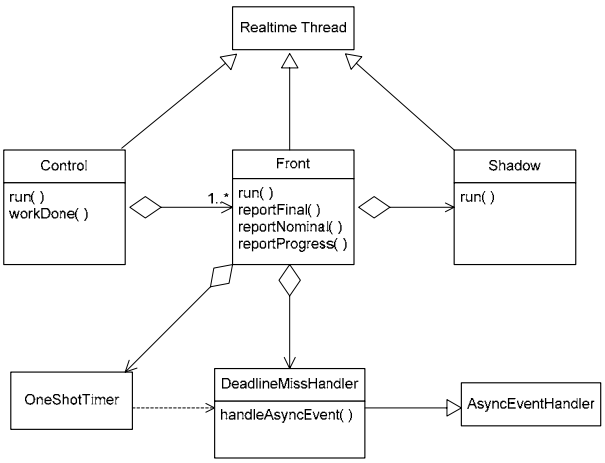

Figure 4. The Shadow pattern class model.

Participants:

- **RealtimeThread**, **OneShotTimer** and **AsyncEventHandler**
  - Java Realtime API Standard classes

- **Control**
  - Creates the Front objects to carry out the time-constrained computations
  - Destroy the Front objects when the computation is completed
- **Front**
  - Creates the Shadow object to carry out the detailed computations
  - Creates the DeadlineMissHandler object
  - Creates the OneShotTimer object with its duration equal to the deadline and associates it with the DeadlineMissHandler
  - Keeps track of updates from the Shadow object
  - Reports either the full result or the nominal result to the Control object when the reportFinal() or reportNominal() method is called
- **Shadow**
  - The reusable component that performs the actual computation
  - Calls the reportFinal() method of the Front object when computation is done
- **DeadlineMissHandler**
  - Used by the OneShotTimer to call the reportNominal() method of the Front object

Collaboration:

Scenario 1. The Shadow object completes its computation before the deadline.

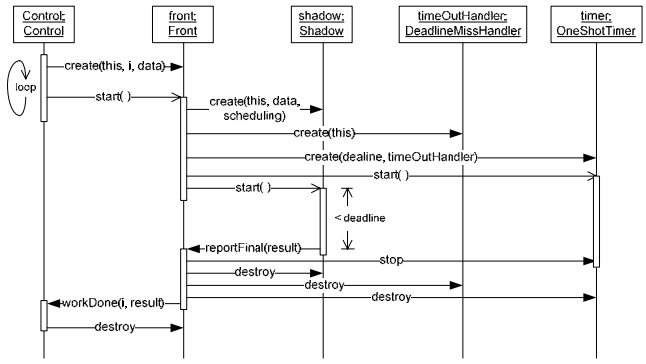

Figure 5. Scenario 1 sequence diagram.

Scenario 2. The Shadow object fails to complete its computation before the deadline.

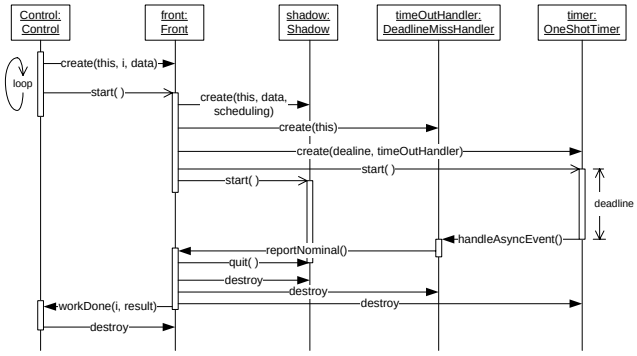

Figure 6. Scenario 2 sequence diagram.

Sample Code:

The Java code shown here sketches the implementation of the Control, Front, Shadow and the DeadlineMissHandler classes in the Shadow pattern.

- **Control**

```
public void run( ) {
  for (int i = 0; i < repeatCnt; i++) {
       DataItem item = new DataItem(i);
       Front front =
              new Front(this, i, item);
       dataStore[i] = front;

       //other bookkeeping tasks
    }

    RelativeTime delay =
                new RelativeTime(50, 0);

    for (int i = 0; i < repeatCnt; i++) {
       dataStore[i].start();
       try {
           RealtimeThread.sleep(delay);
       } catch (InterruptedException e) {
       }
    }
}


public synchronized void workDone(
                  int id, DataItem result) {
    dataStore[id] = null; //remove it, so it
                          //gets garbage
                          //collected

    //other bookkeeping tasks
}
```

- **Front**

```
public void run( ) {
    PriorityParameters scheduling =
       new PriorityParameters(
             PriorityScheduler.
                instance().getMinPriority());
    shadow =  new Shadow(this,
       dataItem, scheduling);
    DeadlineMissHandler timeoutHandler =
       new DeadlineMissHandler(this);
    timer = new OneShotTimer(
       new RelativeTime(
             controller.getDeadline(), 0),
       timeOutHandler);

    timer.start();
    shadow.start();
}

public synchronized void reportFinal(
                          DataItem result ) {
    if (isActive) {
       isActive = false;
       timer.stop(); //we got a full result
                     //from the stateless
                     //discriminator so stop
                     //this OneShotTimer
                     //object
       timer = null;
       shadow = null;
       timeOutHandler = null;

       control.workDone(id, result);
    }
```

```
}

public synchronized void reportNominal( ) {
    if (isActive) {
        isActive = false;
        shadow.quit(); //this kills the shadow
                       //by setting its
                       //'isActive' to false.
        shadow = null;
        timer = null;
        timeOutHandler = null;

        control.workDone(id, nominalResult);
    }
}

public synchronized void reportProgress(
                          DataItem result) {
   //bookkeeping tasks
```

```
}
```

- **Shadow**

```
public void run( ) {
    while (isActive && i < 100) {
         //do work
    }

    front.reportFinal(result);
 }
```

- **DeadlineMissHandler**

```
public void handleAsyncEvent( ) {
    front.reportNominal();
}
```