developer.*
A Web Magazine for Software Developers

The Task Pattern: A Design Pattern for Processing and Monitoring Long-Running Tasks

by Hugo Troche

Abstract

This paper is divided in eight sections: In the first section I examine the problem and motivation for this design pattern. In the second section I explain the pattern in a basic “hello world” manner. Then in the third section I integrate the Task Pattern with the Gang of Four Command, Command Holder and Mediator patterns. In the fourth section I explain how to use the Task Pattern with a worker thread. (The worker thread approach is recommended for production systems.) In the next section the paper explores implementing the pattern with aspects. In the sixth section I present a quick case study of the use of the Task Pattern. In the last two sections the paper does a summary and conclusion and cites references.

1. Problem

Sometimes it is necessary that tasks that take longer than a few seconds—like opening a large file, establishing a network connection, performing a long calculation, etc. —provide feedback to the user and the system [Johnson 2000]. This problem can be solved in many ways, but the Task Pattern presented in this paper provides a proven solution for creating and monitoring long running tasks. With monitor-able tasks the user or the system can know how long a computer task is going to take and then take decisions accordingly.

2. Basic Task Pattern

Idea: The basic Task Pattern involves creating an abstract class, `AbstractTask`, (or its equivalent in languages other than Java) that has methods to set and get the size of the task, the current progress of the task and the current message. Then every task extends `AbstractTask` (`ActualTask`) and overrides the `run` method, which is the method that other objects will call to start the task. Then `ActualTask` informs its parent of the size, progress, and other messages. (See Figure 1 for details.)

 ©2003 by the original author

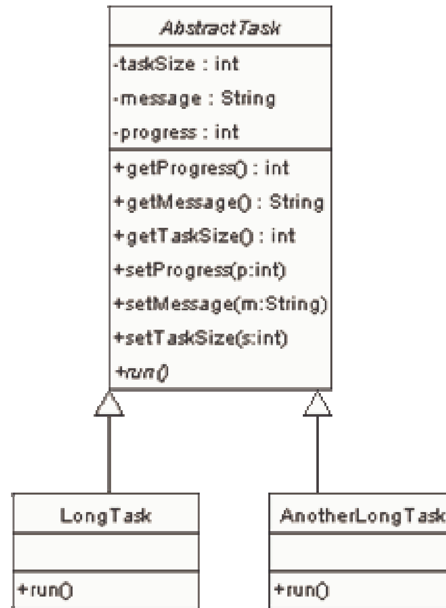

Figure 1: UML diagram for basic `AbstractTask` class and two children

Any other object on another thread can monitor the task by calling the `get` methods for size, progress, and message of the task. That other object has to be on another thread because otherwise the client will not be able to call `getProgress` and `getMessage` until `run` finishes. When `AnyOtherObjectClient` executes `runLongOperationInSeparteThread` it executes `run` in both `LongTask` and `AnotherLongTask`. Then `AnyOtherObjectClient` can monitor the status of `LongTask` and `AnotherLongTask` by calling the `getProgress` and `getMessage` of `AbstractTask`.

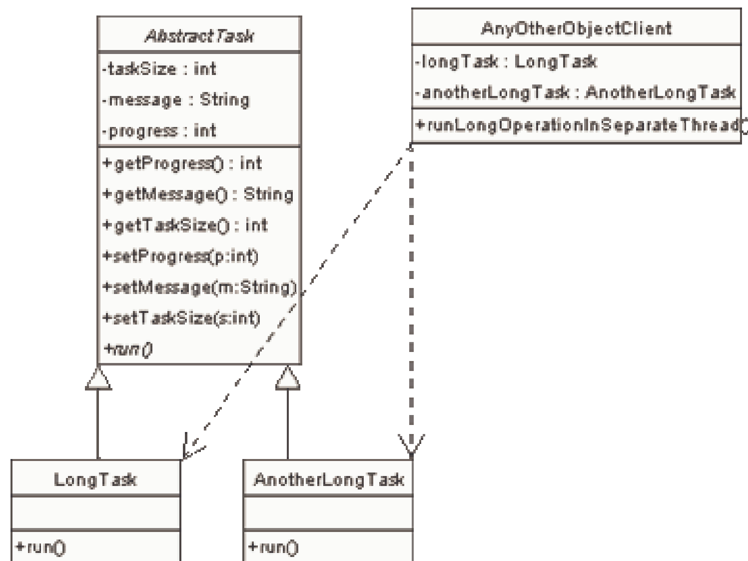

Figure 2: UML class diagram for `ActualTask` and Client

 ©2003 by the original author

Of course, this implementation is inefficient because the client of the task has to constantly monitor the task. It is much better if the task tells its clients when it has changed. That can be achieved by using the Gang of Four Command, Command Holder and Mediator patterns [Gamma, et al 1995].

3. Advanced Task Pattern

As mentioned in the previous section, it is better to use the Task Pattern in combination with the Command, Command Holder, and Mediator patterns [Gamma, et al 2002]. The Command pattern requires creating small "command" objects that derive from a `Command` interface. These small objects perform short actions. I will go over the Command Holder pattern later. The Mediator pattern reduces coupling by keeping track of different objects and establishing how one object should react to an event of another without these objects knowing about each other.

Idea: Create an abstract class (`AbstractTask`) that all tasks will extend. `AbstractTask` registers listeners that want to monitor the task. Every time a new event is fired in `AbstractTask`, `AbstractTask` passes the right command to all its listeners. Create a Mediator class to be the main listener for the events of the abstract task. The Mediator will then do what the Command object passed by `AbstractTask` tells it to do.

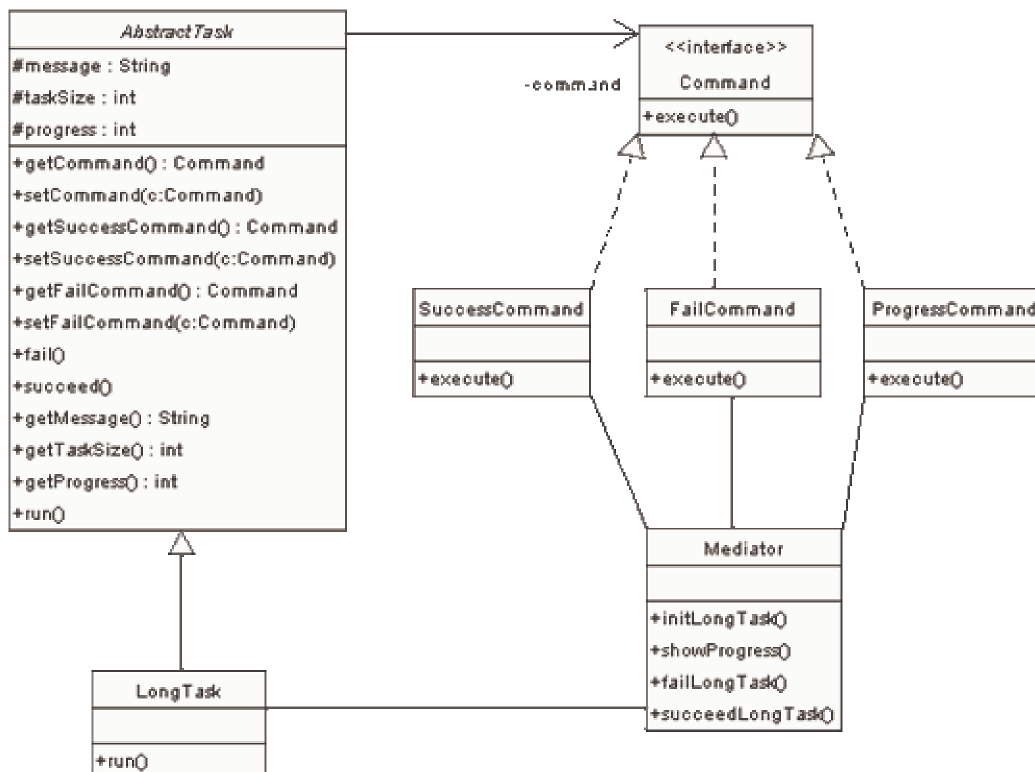

Figure 3: UML class Diagram for Advanced Task Pattern Example

 ©2003 by the original author

The pattern in this case will work as follows: First the Mediator executes `initLongTask` where it will set up the `LongTask` with `SuccessCommand`, `FailCommand`, and `ProgressCommand`, and then it will run the task (See Figure 4). It is important to notice that the implementations of `execute` in `SuccessCommand`, `FailCommand`, and `ProgressCommand` will call the methods `succeedLongTask`, `failLongTask`, and `showProgress` respectively in `Mediator` (See Figure 5 for an example of `SuccessCommand`). This means that we can dynamically specify in `AbstractTask` how the monitoring is done.

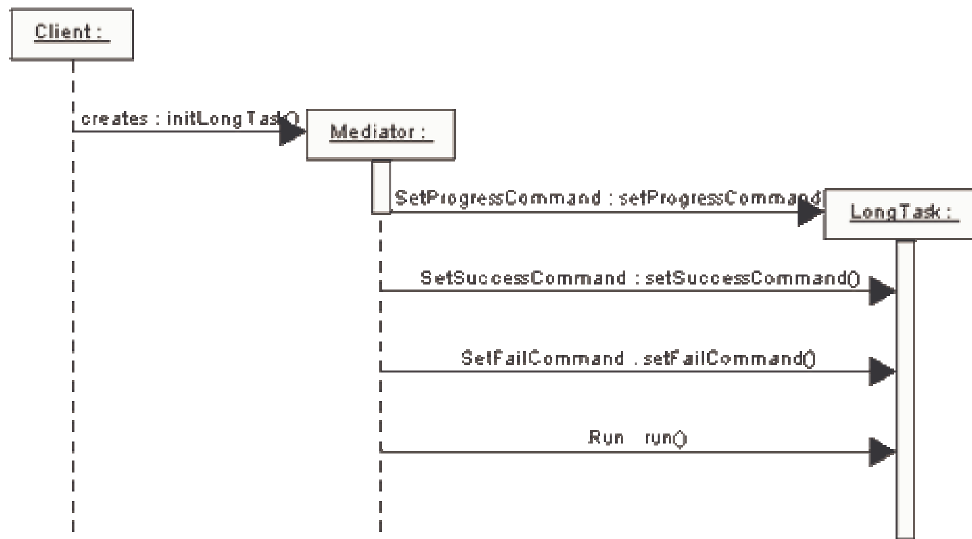

Figure 4: Sequence Diagram for `initLongTask()` in `Mediator`

The logic for when the Command gets called is as follows: every time `LongTask` wants to communicate progress it calls the `execute` method of the Command object returned from calling its parent method `getProgressCommand`. Remember that this Command was set dynamically in the initialization method `initLongTask` in `Mediator`. When `run` is done in `LongTask` it either calls `fail` or `succeed` from its parent. The method `fail` will call the `execute` method of the return Command of `getFailCommand` and the `succeed` method will do the same for the return of `getSucceedCommand`.

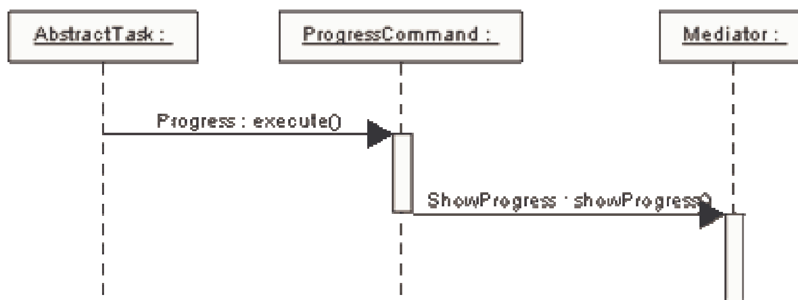

Figure 5: Sequence Diagram for `execute()` in `ProgressCommand`

 ©2003 by the original author

4. Multithreaded Task Pattern

The Task Pattern works best in a multithreaded environment. I picked the Worker Thread pattern [Lea 1999] because it has the most simple implementation. Yet, the Task Pattern can be used in other multithreaded paradigms. For examples of multithreading issues and paradigms check [Magee, Kramer 1999].

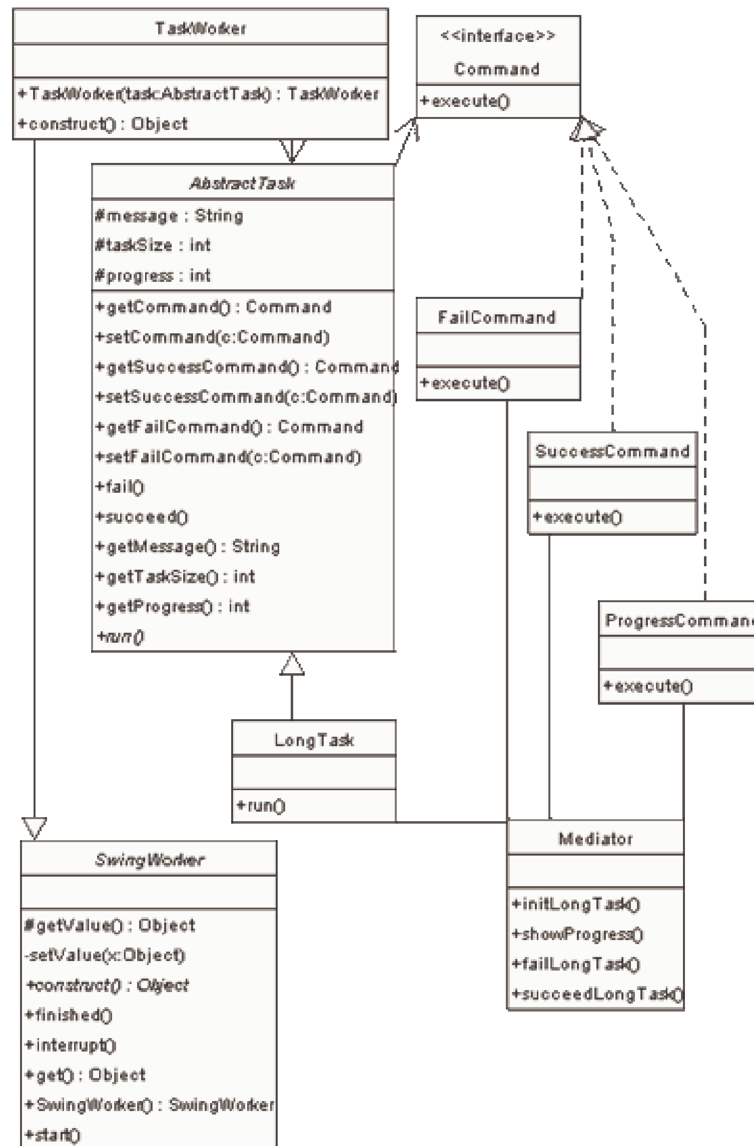

Figure 6: UML class diagram for multithreaded Task Pattern

 ©2003 by the original author

Idea: Create an `AbstractTask` similar to the one described in Section 3. The task uses Commands and Mediators to keep track of the initialization, progress, and termination of the task. Everything until now is similar to Section 3. The difference is that instead of the `Mediator` calling `run`, the `Mediator` will initialize a worker thread that will run the task. This will allow the main thread of execution to maintain control of the application, so the user can use other features of the application while the task is running. What we do is create a `TaskWorker` that will use a `Thread` Singleton [Gamma, et al 1995] to run the contents of the task object. What the mediator does when it wants to run the task is create a new `TaskWorker` with the task that it wants to run. Then it calls `start` of `TaskWorker` and the task runs on a worker thread and communicates with the main thread thru an Observer [Gamma, et al 1995] implemented in the form of the Java event handling mechanism [Horstmann, Cornell 2002].

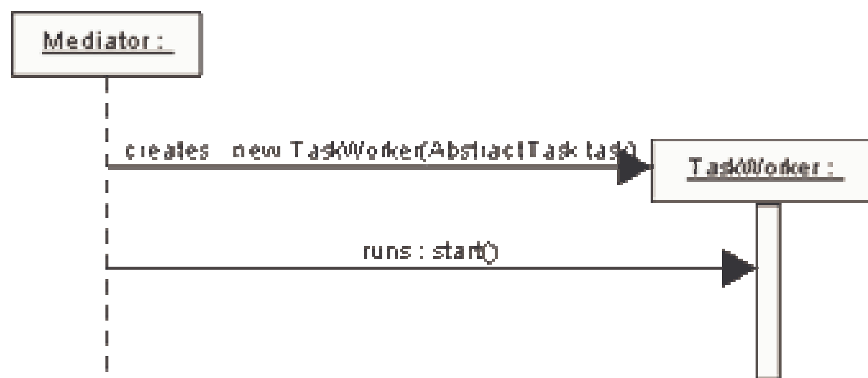

Figure 7: Sequence Diagram for task initialization and to run the task

5. Aspect oriented implementation

Aspect oriented programming presents a new paradigm into software design. Aspects allow the developer to interleave behavior across classes [Kiczales 1996]. The great advantage of aspects is that by using them we can introduce new behavior to classes without changing the classes and without the classes even knowing that the thread of execution will be different. A good source in practical aspect programming is [Laddad 2003].

There has been research in implementing object oriented design patterns with aspects. For example [Hannemann, Kiczales 2002] implemented all of the Gang of Four design patterns [Gamma, et al 1995] with aspects and analyzed performance and cohesion of the design. Another good source of research in this area is [Clarke, Walker 2001], which analyzes reusability of aspects thru patterns.

The Task design pattern can be implemented with aspects. Instead of using the event listening infrastructure [Horstmann, Cornell 2002] the task should use an observer pattern implemented with aspects as prescribed in [Hannemann, Kiczales 2002]. This will reduce coupling in the system since event generators and event listeners don’t need to know about each other. On the other hand, the task will have to call a method inside its own `run`

 ©2003 by the original author

method every time it wants to generate a progress report. That method will be the joint point in the aspect monitoring the task. So, we could implement the Mediator as an aspect and reduce coupling.

6. Case Study

I have used and implemented the Task Pattern a number of times. This is a case study of using the Task Pattern for getting, saving and adding data to and from a web service. All the tasks use a StatusPanel [Horstmann, Cornell 2002] to show its status.

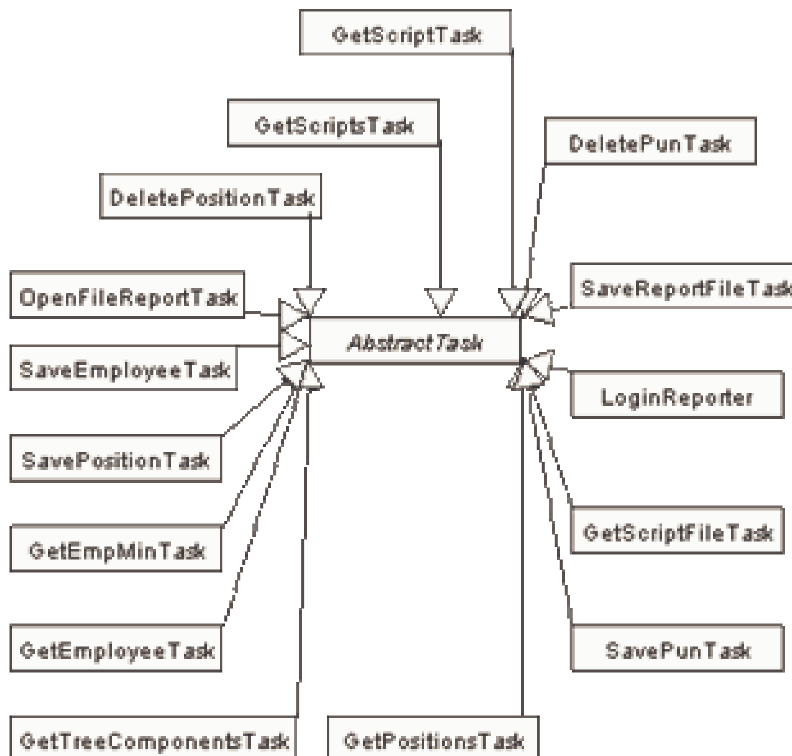

Figure 8: UML class diagram of all the tasks use in Framework application

I used the Task Pattern for an application called Framework. Framework is 3-tier application to manage human resources information. (A good introductory resource for n-tier applications is [Edwards 1999].) The users can search for employees, add and update records, and also delete records. The application also has other reporting and querying features. In Framework every feature that needs to talk to the application server uses the Task Pattern. Each one of these tasks is linked to a particular Mediator with Commands. In this way the tasks can be monitored.

 ©2003 by the original author

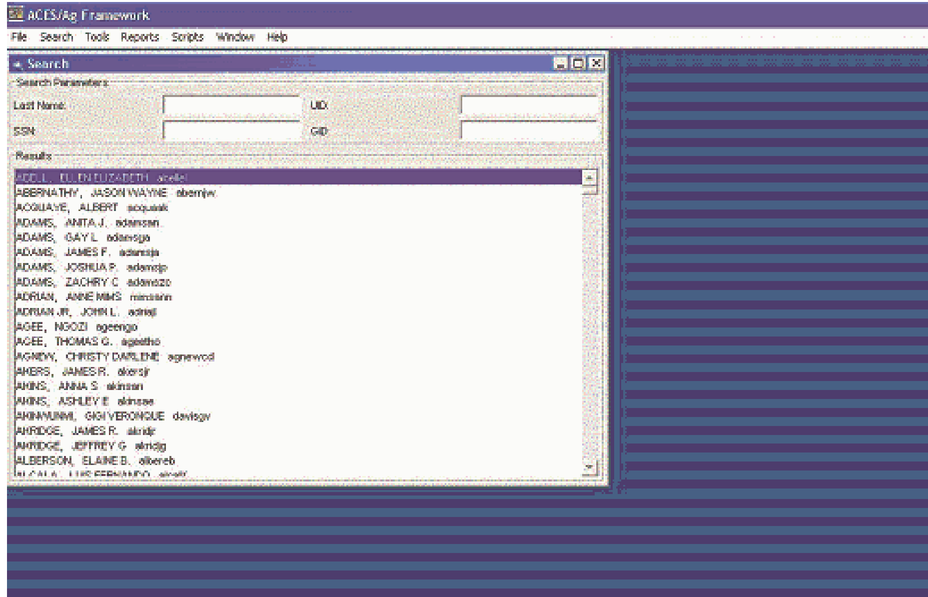

Figure 9: Screen shot of framework

For example, in Figure 10 we can see how `PositionMediator` uses `GetPositionTask` when it wants to get new position information from the application server. So, when `loadPosition` in `PositionMediator` gets called, `PositionMediator` creates a new `TaskWorker` (as explained in Section 4) and then runs `GetPositionTask` in that `TaskWorker`. This prevents the user does from losing control of the application when the system is getting the data for a new position.

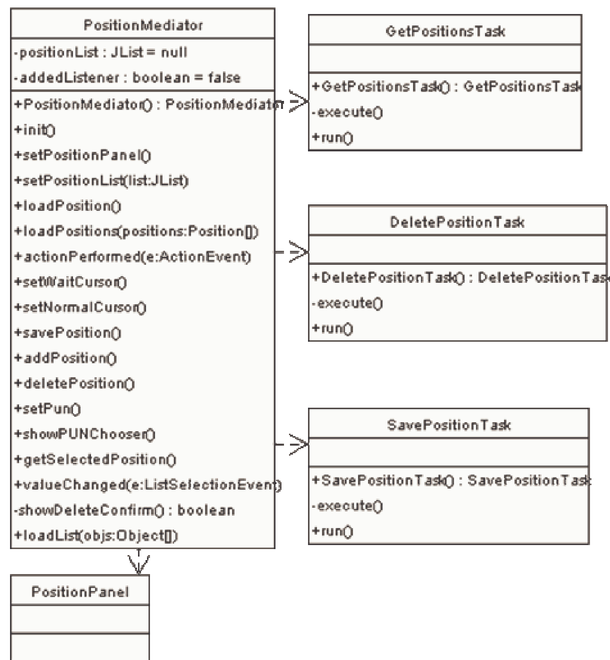

Figure 10: UML class diagram of `PositionMediator` and its tasks

 ©2003 by the original author

The same holds true when deleting a position or saving a position (see Figure 10). One important point to consider is that the tasks in this case are Command Holders [Cooper 2000]. A Command Holders, as its name implies. holds Commands and passes the Commands to the Mediator for execution at the correct time. For more information on Command Holders please see [Cooper 2000].

The same principle is applied to other functionality of Framework. Every time the application has to call the application server, I used the Task design pattern. Doing this gives Framework a consistent texture [Ran 2001], and once a developer understands how one feature is implemented, she will easily understand how all the other similar features are implemented. (A good resource for implementation guidelines for design patterns is [Beck, et al 1996].)

7. Conclusion

In this paper I discussed the Task design pattern. This pattern solves the problem of how to structure long tasks that require monitoring.

I explored the various flavors of the Task design pattern. I first started with a simple implementation of it, and then moved to an implementation with Commands. After that I showed a multithreaded implementation. Then I explored the possibility of implementing this pattern with aspects. I finished the paper with a case study from my work experience.

References

[Gamma, et al 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. 1995. *Design Patterns*. Addison-Wesley.

[Cooper 2000] James W. Cooper. 2000. *Java Design Patterns: A Tutorial*. Addison-Wesley

[Lea 1999] Doug Lea. 1999. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley.

[Laddad 2003] Ramnivas Laddad. 2003. *AspectJ in Action: Practical Aspect Oriented Programming*. Mannings Publications Company.

[Horstmann, Cornell 2002] Cay S. Horstmann, Gary Cornell. 2002. *Core Java 2, Volume I: Fundamentals* (6th edition). Prentice Hall

[Magee, Kramer 1999] Jeff Magee, Jeff Krammer. 1999. *Concurrency: State Models & Java Programs*. John Wiley & Sons.

[Edwards 1999] Jeri Edwards. 1999. *3-Tier Server/Client at Work*. John Wiley & Sons.

[Kickzales 1996] G. Kiczales. 1996. *Aspect-Oriented Programming*. ACM Computing Survey. Volume 4: 157.

 ©2003 by the original author

[Hannemann, Kiczales 2002] Jan Hannemann, Gregor Kiczales. 2002. “Design Pattern Implementation in Java and AspectJ”. *Proceedings of the 17th ACM SIGPLAN Conference* (November 04-08). Seattle, Washington.

[Agerbo, Cornilis 1998] Ellen Agerbo, Aino Cornilis. “How to Preserve the Benefits of Design Patterns”. 1998. ACM SIGPLAN Notices, *Proceedings of the 13th ACM SIGPLAN Conference on Object Oriented Programming* (October 18-22). Vancouver, British Columbia, Canada.

[Johnson 2000] Jeff Johnson. *GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*. Morgan Kaufmann.

[Ran 2001] Alexander Ran. 2001. “Fundamental Concepts for Practical Software Architecture”. *Proceedings of the 8th European Software Engineering Conference* (September 10-14). Vienna, Austria.

[Clarke, Walker 2001] Siobhan Clarke, Robert J. Walker. 2001. “Composition Patterns: An Approach to Designing Reusable Aspects”. *Proceedings of the 14th International Conference on Software Engineering* (May 12-19). Toronto, Ontario, Canada.

[Borchers 2000] Jan O. Borchers. 2000. “A Pattern Approach to Interaction Design”. *Proceedings of the Conference on Designing Interactive Systems* (August 17-19). New York, New York.

[Beck, et al 1996] Kent Beck, Ron Crocker, Gerard Meszaros, John Vlissides, James O. Coplien, Lutz Dominick, Frances Paulisch. 1996. “Industrial Experience with Design Patterns”. *Proceedings of the 18th International Conference on Software Engineering* (March 25-29). Berlin, Germany.

###

Hugo Troche is a software engineer living in Auburn, Alabama, USA. He is originally from Asuncion, Paraguay and has been living in Alabama for four years. He earned his undergraduate degree in Computer Science from Auburn University (Magna Cum Laude) in 2002 and finished his Master's degree in 2004. He is also a software consultant and contractor specializing in web services and n-tier business systems. Hugo can be reached at `trochph@auburn.edu`.

 ©2003 by the original author